MASTER IN COMPUTER SCIENCE

Contrôle d'admission du serveur d'applications IMS d'Alcatel-Lucent
Rétro-ingénierie et amélioration

Bouhy, Michaël

Award date:
2010

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'informatique

Année académique 2009 - 2010

Contrôle d'admission du serveur d'applications IMS d'Alcatel-Lucent.

Rétro-ingénierie et amélioration.

Michaël Bouhy

Mémoire présenté en vue de l'obtention du grade de Master en Informatique

Abstract

This thesis presents a retro-engineering and an improvement of the admission control of the platform of Alcatel-Lucent (ASR). The admission control is managed by two algorithms : the flow control algorithm and the global load estimation algorithm of the system. On the one hand, both algorithms were improved. On the other hand, their behaviour has been formalised thanks to two models.

Reliability, maintainability and good performance of the ASR require a redundant architecture in components, a division of these into containers, the implementation of strategies like the respect of the pattern Reactor/Proactor and the massive parallelisation of the treatment of the messages. Moreover, policy of the development team of Alcatel-Lucent consists in developing generic functionalities including the admission control. These principles complicates the improvement of a global load estimation algorithm of the system. A solution is presented in detail and a case study shows that the admission control is improved.

Résumé

Ce mémoire présente la rétro-ingénierie et une amélioration du contrôle d'admission. Le contrôle d'admission de la plateforme d'Alcatel-Lucent (ASR) est géré par deux algorithmes : l'algorithme du contrôle de flux et l'algorithme de l'évaluation de la charge globale du système. D'une part, l'algorithme de contrôle de flux et surtout l'évaluation de la charge globale du système ont été améliorés. D'autres parts, leur comportement est formalisé grâce à deux modèles.

La fiabilité, la maintenabilité et les bonnes performances de l'ASR nécessitent une architecture redondante en composants, une découpe de ceux-ci en conteneurs, la mise en oeuvre de stratégies comme le respect du pattern Reactor/Proactor et la parallélisation massive du traitement des messages. La ligne de conduite de l'équipe de développement d'Alcatel-Lucent consiste à développer des fonctionnalités génériques dont le contrôle d'admission. Ces principes ont compliqué l'amélioration et la mise au point d'un algorithme d'évaluation de la charge globale du système. Une solution est présentée en détail et une étude de cas montre que le contrôle d'admission est amélioré.

Remerciements

Au terme de cette année de travail au sein de deux équipes compétentes, en Belgique et en France, je tiens à remercier toutes les personnes qui m'ont permis de mener à bien ce mémoire. Je remercie particulièrement mes superviseurs, Laurent Schumacher et Dimitri Tombroff.

À Namur, Laurent Schumacher et Antoine Roly ont été patients pendant la lecture de mes rapports et pendant la correction du mémoire, car mes idées étaient quelques fois confuses et difficiles à expliquer, mais les discussions fructueuses que nous avons eues pendant nos réunions avec Marie-Ange Remiche m'ont beaucoup aidé.

Laurent Schumacher a rendu possible le stage à Alcatel-Lucent avec Dimitri Tombroff, leader d'une équipe de Recherche et Développement à Paris. J'ai eu la chance d'intégrer, pendant cinq mois à Paris, cette équipe dynamique. J'ai beaucoup apprécié travailler avec Dimitri : j'ai été marqué par sa volonté de formaliser rigoureusement certains processus très complexes, sa concentration sur la cause d'un problème et non sur ses symptômes et sa perpétuelle remise en question. J'espère acquérir dans ma future carrière la même méthodologie pour traiter les problèmes.

Dans cette équipe, je remercie Arjun Penday et Pierre De Rop, ingénieurs du logiciel. Arjun, un grand voyageur, m'a toujours donné des explications claires et pédagogiques pour me dépanner. Pierre m'a rendu de fiers services, spécialement lorsque plus aucune commande n'arrivait à débloquer des serveurs.

Merci à Jean-Francois Martin, manager de projet dans une autre équipe avec laquelle nous avons travaillé, nous avons beaucoup réfléchi et testé des solutions à nos problèmes communs dans une très bonne ambiance. Jean-François m'a également fourni les outils nécessaires pour procéder à des tests et m'en a expliqué leur fonctionnement.

Je remercie également la Fondation Biermans-Lapôtre, la maison des étudiants belges et luxembourgeois, pour m'avoir logé à la Cité Universitaire de Paris, car elle m'a permis de rencontrer énormément de personnes et elle a contribué à rendre mon séjour en France agréable et unique.

Je tiens à remercier ma famille et Tina pour leur soutien, ainsi que mes camarades de classe avec lesquels nous nous sommes motivés dans cette dernière ligne droite.

Je remercie enfin Anne-Marie Breny, Jean-Marie Jacquet et Wim Vanhoof qui ont organisé sur mesure mon passage de la faculté des Sciences à la Faculté d'Informatique. Je suis très reconnaissant pour l'excellente formation que j'ai reçue à l'Université de Namur.

Table des matières

Table des figures

Liste des acronymes

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
ASR	Application Server Runtime
CPU	Central Processing Unit
DNS	Domain Name System
FAQ	Frequently Asked Questions
FIFO	First In, First Out
HA	High Availability
HTTP	Hypertext Transfer Protocol
ID	Identifiant
IETF	Internet Engineering Task Force
IP	Internet Protocol
Ipsec	Internet Protocol Security
JMX	Java Management Extensions
MSRP	Message Session Relay Protocol
MUX	Multiplexing protocol
RADIUS	Remote Authentication Dial-In User Service
RTP	Real-time Transport Protocol
RTT	Round-Trip delay Time
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SMPP	Short Message Peer-to-Peer
SMTP	Simple Mail Transfer Protocol
SQL	Structured Query Language
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UAC	User Agent Client
UAS	User Agent Server
UDP	User Datagram Protocol
XCAP	XML Configuration Access Protocol
XDM	XML Document Management
XDMS	XML Document Management Server
XML	Extensible Markup Language

Chapitre 1

Avant-Propos

Le contrôle de charge d'un cluster d'application SIP (Session Initiation Protocol) est un défi, car il possède de fortes implications avec les stratégies de partage de charge, avec les exigences de haute disponibilité et avec les caractéristiques des applications (allant du simple serveur ou proxy SIP aux applications plus lourdes combinant différents protocoles). Un contrôle de charge parfait règle tous ces problèmes, utilise un minimum de paramétrisation et ne requiert pas de connaissances applicatives.

Théoriquement, la plateforme de télécommunication d'Alcatel-Lucent offre l'état de l'art du contrôle d'admission. Celui-ci a été conçu pour gérer dynamiquement des clusters hétérogènes [1] afin de gérer des cas de pannes matérielles, de redémarrage d'application et de redimensionnement de clusters. Mais il fait encore l'objet de recherches, en particulier dans la différenciation des messages (INVITE, SUBSCRIBE, REGISTER, PUBLISH, etc.) et leur association à un poids spécifique (coût CPU, coût mémoire, temps de réponse, etc.).

L'objectif de ce mémoire est double. Il présente une rétro-ingénierie et une amélioration du contrôle d'admission développé à Alcatel-Lucent depuis plus de dix ans. Il s'agissait d'une part d'améliorer l'algorithme de contrôle de flux et surtout l'évaluation de la charge globale du système et d'autre part de formaliser son comportement.

Les algorithmes de gestion de flux, de partage de charge et d'évaluation de la charge globale du système sont des algorithmes distribués qui utilisent de nombreuses informations de monitoring de la plateforme. Il est par conséquent indispensable de présenter certains aspects du serveur d'applications d'Alcatel-Lucent (voir le chapitre 2).

Les objectifs des algorithmes sont définis en détail dans le chapitre 3 avant de présenter leur implémentation et leurs évolutions dans le chapitre 4.

Deux modèles analytiques du serveur d'application ont été construits et sont présentés dans le chapitre 5. Ils montrent en quoi la modification des nombreux paramètres n'a qu'une influence limitée sur le comportement de la plateforme en exécution. Nous présentons enfin une étude de cas dans le chapitre 6 pour illustrer le bon comportement du système.

Un outil a été spécifiquement développé pour exploiter au mieux un service de monitoring de la plateforme. Il a été très utile pour mener notre réflexion et il est désormais intégré dans la version de la plateforme en production. Son utilisation et son fonctionnement sont présentés dans le chapitre 8.3.

1. Hétérogène d'un point de vue matériel. Des composants identiques de la plateforme n'ont pas forcément droit à des ressources équitables.

Chapitre 2

La présentation du serveur d'applications 5450 d'Alcatel-Lucent

Les services de communication multimédia sont souvent employés comme des services de médiation entre différentes entités de réseaux différents pour transférer du texte, des images ou des sons. Pour permettre ces échanges, des sessions sont établies, modifiées et supprimées entre les utilisateurs à l'aide du protocole SIP. Le serveur d'application (ASR) gère l'ensemble des communications et il est naturellement orienté vers la signalisation et le proxying.

L'ASR est un environnement de développement, de déploiement et d'exécution de services de télécommunication dont l'administration est permise grâce à une infrastructure dédiée. Les protocoles nécessaires à l'exécution de services de communication, SIP et Diameter, HTTP, RADIUS, SMTP, SMPP, etc., sont gérés par l'ASR comme le montre la figure 2.1.

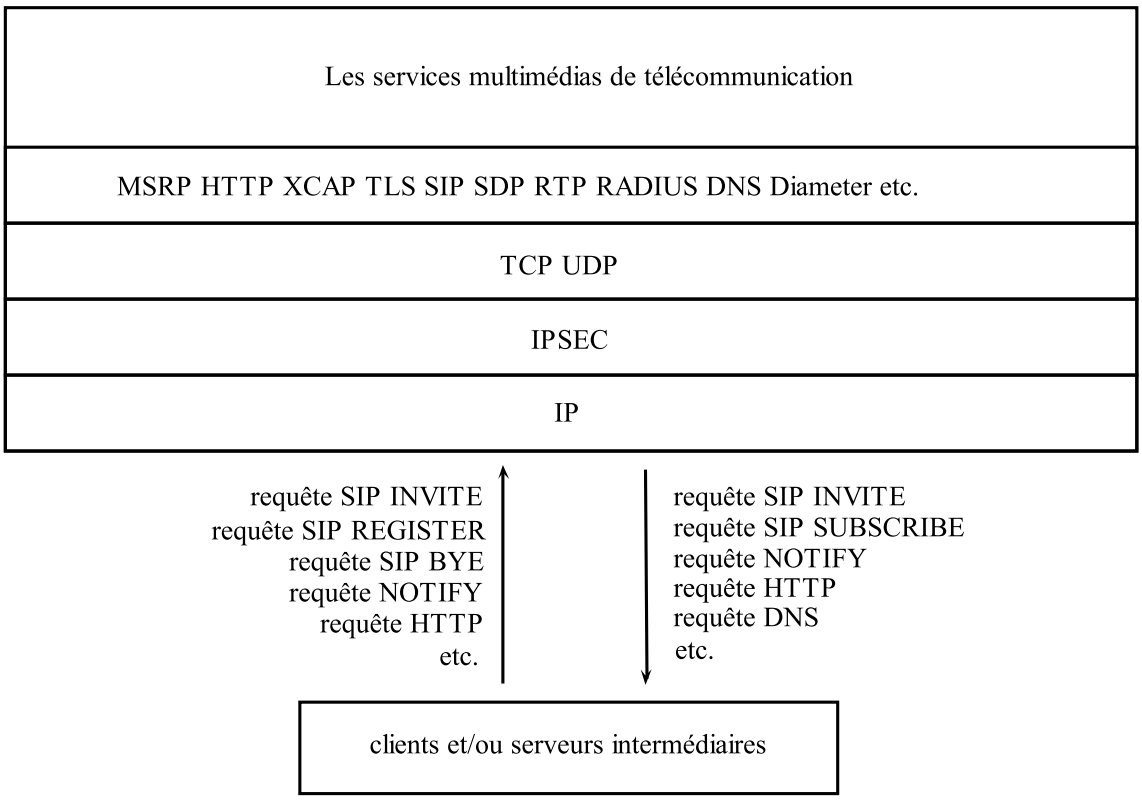


Figure 2.1 – La prise en charge des protocoles de communication dans l'ASR.

Ce chapitre présente le fonctionnement et les principes architecturaux de la plateforme, car ils sont intimement liés au contrôle d'admission de celle-ci. Le contenu du chapitre est basé sur quatre documents d'Alcatel-Lucent, [8, 10, 9, 13], mais il est complétée par la littérature et des prises de notes et des observations personnelles.

2.0.1 La haute disponibilité et la fiabilité de l'ASR

Un serveur d'applications doit pouvoir exécuter un nombre important d'applications et servir un nombre très important d'utilisateurs (quelques millions) avec une très grande disponibilité (la règle des 5 neufs, c'est-à-dire 99,999% de disponibilité par an). L'architecture de l'ASR a été conçue pour partager naturellement la charge applicative et adapter facilement le déploiement pour suivre l'évolution de la charge.

La figure 2.1 est un bloc qui représente l'entièreté de l'ASR, en réalité ce bloc n'est pas d'un seul tenant, mais il est séparé en deux composants, l'agent et l'io_handler (voir la figure 2.2).

La couche supérieure traite proprement dit le trafic. Elle inclut une pile protocolaire complète pour décapsuler et encapsuler les données des messages en fonction des protocoles utilisés. Cette couche exécute les services multimédias de télécommunication. Ils sont écrits en Java, car son utilisation pour implémenter un serveur offre de nombreux avantages : la machine virtuelle de Java offre un mécanisme de gestion automatique de la mémoire, elle est multiplateforme et contrairement aux idées reçues, l'exécution d'une application Java n'est pas plus lente à l'exécution qu'un programme écrit C. Par ailleurs, la machine virtuelle de Java offre en permanence l'état de l'art des algorithmes et permet même d'optimiser en temps réel une application à l'exécution [23][11].

La couche la plus basse intercepte le trafic IP grâce à un composant dédié à un protocole, il est appelé io_handler. Il gère les entrées et les sorties des sockets et réalise un parsage limité des datagrammes dans l'objectif de déterminer si un flux est nouveau ou s'il est possible de le corréler avec un existant. Un flux correspond à n'importe quel échange d'une communication entre un message de requête initiale et un message de réponse finale associé. Les messages participants à une même conversation sont appelés messages subséquents et doivent être traités par le même agent. L'io_handler ne traite pas les messages, mais les oriente vers les agents. Cette couche est écrite en C.

La communication entre les io_handlers et les agents est orchestrée via le protocole MUX, car c'est une communication interne à l'ASR. La séparation de l'ASR en différents io_handlers et agents permet de rendre le système robuste par de la redondance, mais l'oblige à contrôler les flux et le partage de la charge applicative.

2.0.1.1 La redondance

L'ASR doit être hautement disponible alors que les erreurs matérielles ou les pannes sont fréquents. Pour diminuer ce risque d'interruption de service, les agents sont déployés sur des machines différentes permettant de réaliser de la redondance. L'ASR est par conséquent fortement disponible (HA) [1], car si une machine rencontre un problème matériel alors tous les agents hébergés disparaissent et ne peuvent plus traiter le trafic entrant. Le trafic est alors redirigé vers les autres agents sur lesquels les sessions auront été dupliquées.

Cette duplication est réalisée à l'aide de transactions distribuées de session (voir section 2.2.3). Ce mécanisme permet à un agent de prendre en charge une session d'un autre agent si celui-ci vient à disparaître. Par principe, une session est dupliquée en permanence sur un minimum de deux hôtes différents, un hôte maître et un hôte esclave. L'état des sessions sur les hôtes esclaves est continuellement synchronisé sur les sessions des hôtes maîtres.

Les applications sont déployées dans un groupe applicatif qui définit le périmètre dans lequel la redondance des sessions est activée.

Un groupe est une collection de composants proxy qui réalise une tâche donnée. La plupart des composants de l'ASR sont de type proxy. [2] Chaque application déployée a besoin de plusieurs composants proxy pour fonctionner. Les deux grandes catégories sont les io_handlers (il y en a un pour chaque protocole, HTTP, SIP, etc.) et les agents.

Tous les agents d'un groupe exécutent la même liste d'applications proxy, supportent les mêmes protocoles, sont connectés aux mêmes io_handler et possèdent la même configuration.

1. high availability
2. D'autres composants de l'ASR sont de type serveur. On rencontrera la JMXGateway, la Fastcache, Syslog NG, etc.

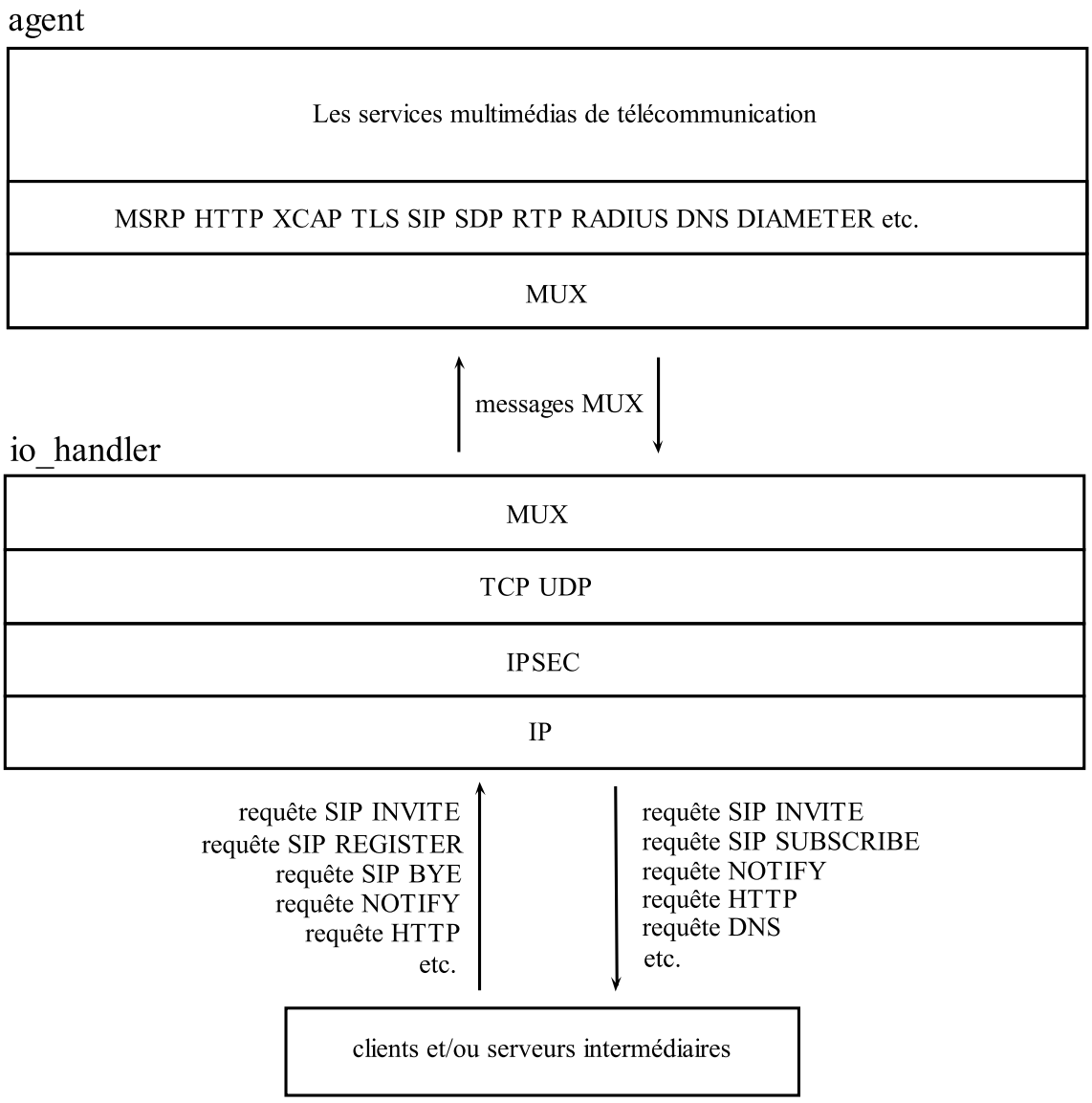


Figure 2.2 – Les servlets et les proxylets.

2.0.1.2 Le contrôle du partage de la charge

Les io_handlers dirigent le trafic vers plusieurs agents identiques. Cette redirection des messages ne demande pas beaucoup de ressources CPU car le parsage des messages est très limité. Les agents sont plus fortement affectés par l'arrivée de messages à cause du traitement applicatif, mais ils se partagent la charge. Au fur et à mesure que les besoins augmentent, de nouveaux agents peuvent être dynamiquement ajoutés à la plateforme pour accroître ses capacités.

Le contrôle du partage de la charge est placé dans une couche basse de l'ASR, en dessous de la couche protocolaire et applicative. Il est multiprotocole et générique.

2.0.1.3 Le contrôle d'admission

La saturation du système est évitée grâce à un mécanisme qui évite indépendamment la saturation de chaque agent grâce au système de contrôle d'admission. Il permet de limiter la quantité de trafic échangé entre un io_handler et un agent.

Comme pour le contrôle du partage de la charge, le contrôle d'admission est placé dans une couche basse de l'ASR, en dessous de la couche protocolaire et applicative. Il est multiprotocole et générique.

2.1 La gestion de la complexité dans l'agent

La maintenabilité de l'agent est assurée par son organisation en conteneurs et enablers. La finalité d'un agent est d'exécuter des applications, celles-ci utilisent un ensemble de fonctions offertes, soit par le conteneur générique, soit par le conteneur spécifique à un protocole.

2.1.1 Les conteneurs

L'écriture *from scratch* d'applications supportant la mise à l'échelle et la haute disponibilité est complexe, car il faut gérer la gestion des états, des problèmes liés au multi-threading, aux réservations de ressources et d'autres détails complexes. La décomposition de l'agent permet de gérer cette complexité et de la cacher aux développeurs d'applications en leur offrant des conteneurs.

La notion de conteneur est populaire en Java. De nombreuses API sont disponibles telles que l'API Servlet [3]. Les proxylets sont une combinaison de services qui peuvent contrôler et modifier le flux d'information échangé entre un client et un serveur. Par exemple, un proxylet peut charger une requête, interdire l'accès à un service, modifier la réponse d'un serveur, etc.

L'ASR est composé de deux types de conteneurs, les génériques et les protocolaires.

2.1.1.1 Les conteneurs génériques

Ce type de conteneur offre un ensemble de services génériques (indépendant des protocoles de communication) dont deux très importants, la gestion du protocole de multiplexage des sockets (MUX) et la gestion des sessions.

MUX est un service implémentant un protocole de transport d'information multiplexé interne à l'ASR (entre un agent et un io_handler). MUX permet d'optimiser les performances de la plateforme, car son principe est de multiplexer plusieurs connexions entre deux applications dans un seul socket TCP. Les applications évitent ainsi d'ouvrir autant de sockets TCP qu'il n'y a de connexions. Le nombre d'appels système d'écriture/lecture est fortement diminué, car un appel permet de servir plusieurs connexions à la fois dans un seul socket.

La gestion des sessions est un service fournissant aux utilisateurs un moyen transactionnel de stockage des sessions en mémoire. Ce service permet aux développeurs de créer des sessions spécifiques à leur application.

Une session est considérée comme un échange de données entre une association de participants [18]. Il est possible de créer, détruire, modifier une session, mais aussi d'apposer un listener sur ces actions pour être à tout moment prévenu d'un changement d'état.

2.1.1.2 Les conteneurs protocolaires

Les conteneurs protocolaires complètent les conteneurs génériques avec un protocole de communication. La plupart des conteneurs protocolaires Java sont dérivés de l'interface Proxylet ou Servlet (par exemple HttpServlet [1]) et leur succession définit une application (voir la figure 2.3).

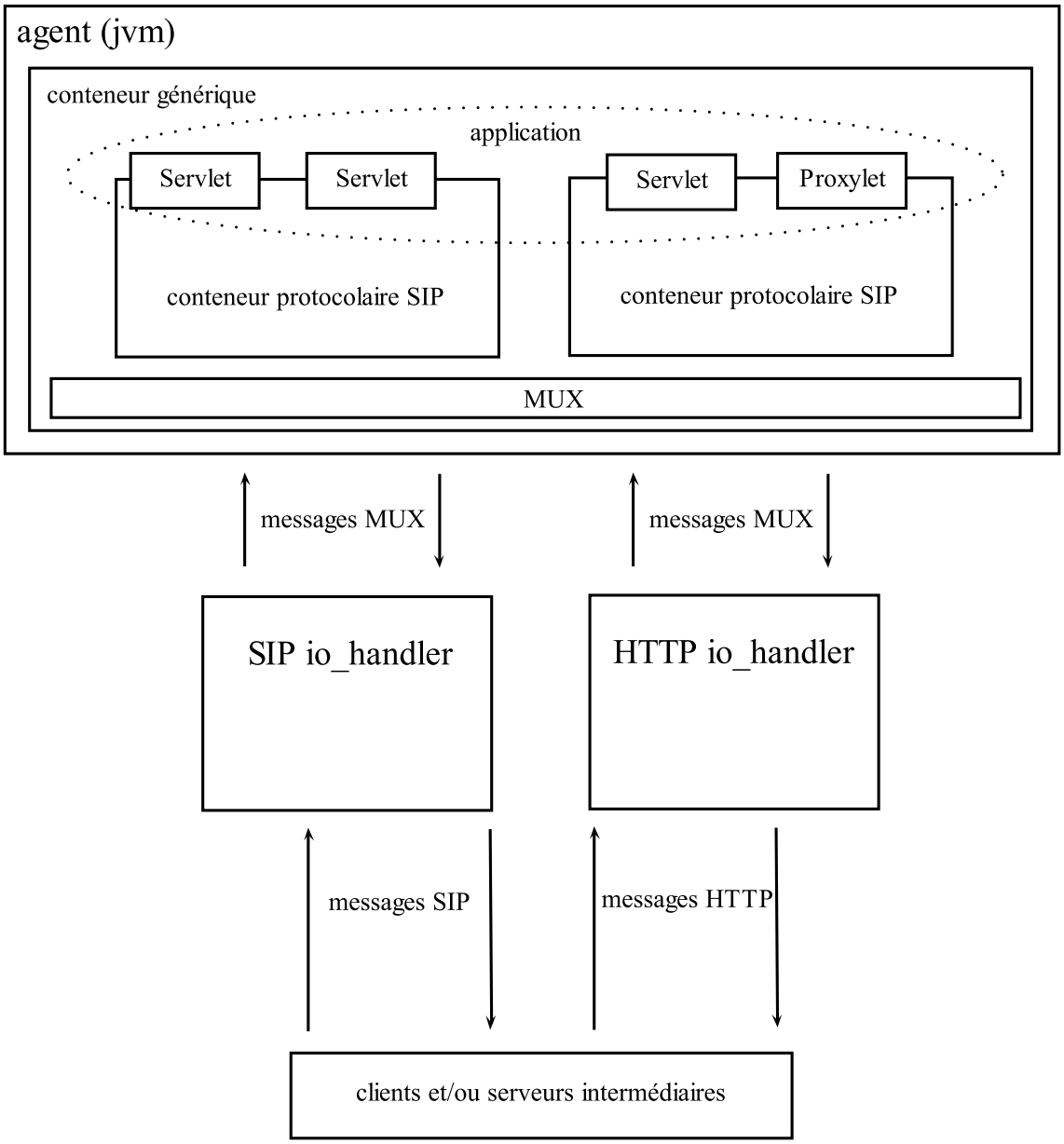


Figure 2.3 – L'organisation de l'agent en conteneurs génériques et protocolaires.

Le conteneur protocolaire SIP Servlet intègre une pile SIP et offre une API SIP Servlet aux applications. Les fonctions des différentes couches sont :

1. Le transport SIP utilise l'API MUX pour gérer les émissions et les réceptions de messages SIP.
2. Le parseur syntaxique SIP parse chaque message SIP entrant.
3. La couche des transactions SIP [21]. Une transaction SIP est une séquence de messages SIP échangés entre les différents éléments du réseau SIP. Une transaction consiste en l'envoi d'une requête et en la réception de toutes les réponses associées à cette requête [21].
4. Les dialogues SIP sont gérés par la couche SIP Servlet. Un dialogue représente une relation SIP pair à pair entre

deux agents. Un dialogue est identifié grâce à un call-ID, un From tag et un To tag. Les messages qui possèdent les mêmes valeurs de ces identificateurs appartiennent au même dialogue [19].

2.1.2 Les enablers

Les possibilités offertes par les conteneurs permettent de simplifier le développement d'applications en cachant la complexité bas-niveau, mais les API ne sont pas encore assez abstraites. Les enablers sont implémentés au-dessus de cette couche d'abstraction. Ils permettent aux développeurs d'accélérer le développement d'applications haut niveau comme une messagerie instantanée. Un enabler est un composant qui offre une interface standard à des services internes ou externes à valeur ajoutée. [3]

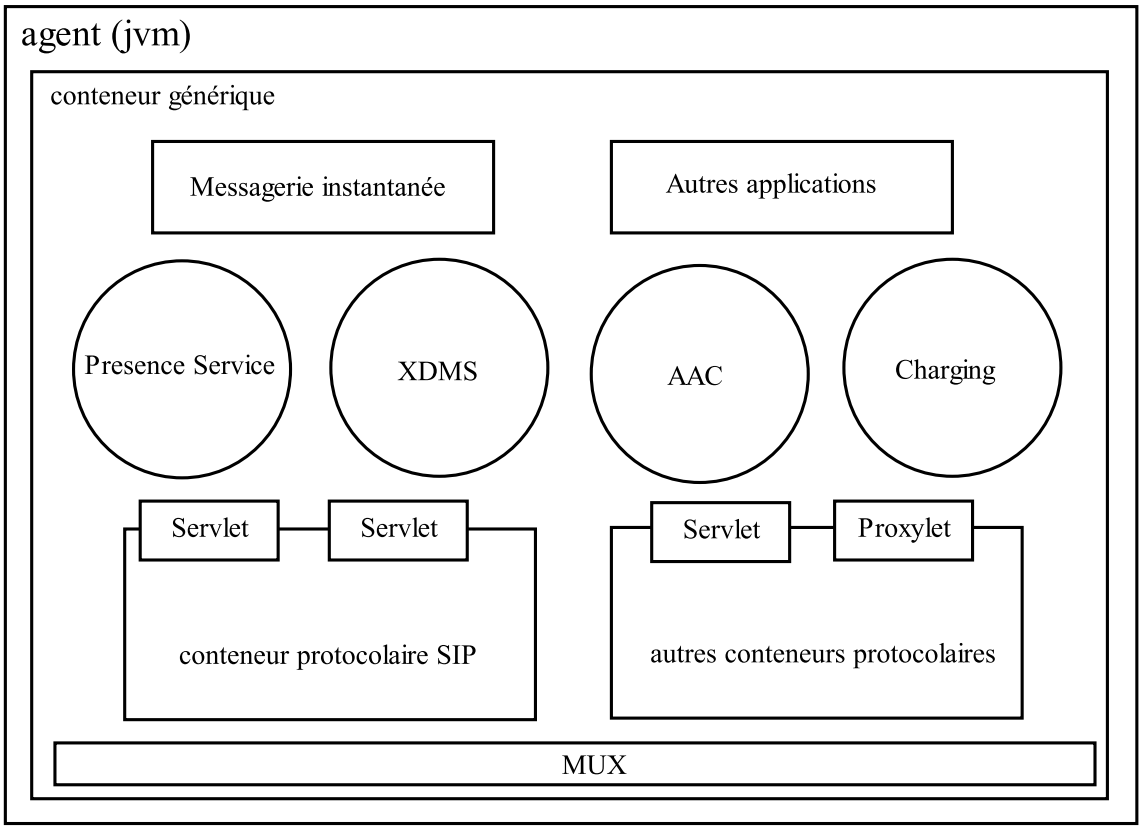


Figure 2.4 – Les enablers.

La figure 2.4 présente les principaux enablers de l'ASR. Le premier est l'enabler d'authentification et de contrôle d'accès, le deuxième est le service Présence, le troisième est le système de gestion de document XML (XDMS) et enfin le quatrième est l'enabler de paiement. Seuls le service XDMS et le service Présence ont besoin d'être rapidement exposés dans ce travail.

2.1.2.1 Le service Présence

La gestion des contacts et leur présence sur le réseau sont souvent présentées en trois avancées majeures par Alcatel-Lucent. La vision d'hier, d'aujourd'hui et de demain. La vision d'hier présente une communication initiée à partir de dispositifs utilisant des carnets d'adresses. Les données résidaient dans les dispositifs de communications des utilisateurs finaux.

La vision d'aujourd'hui de la communication est d'utiliser des informations de présence standardisées. Les utilisateurs finaux ont la possibilité d'accéder et de modifier leur liste de contacts à partir de n'importe quel dispositif de messagerie.

3. c'est-à-dire qu'on peut vendre

Les informations résident sur Internet, mais ne sont pas encore gérées par les opérateurs eux-mêmes.

La vision de demain intègre les opérateurs en utilisant le service Présence. Les utilisateurs auront l'occasion d'accéder et de modifier leur liste de contacts à partir d'une grande variété d'applications de communication. L'opérateur aura la possibilité d'offrir une liste de contacts basée sur le réseau.

2.1.2.2 Le service XDMS

Cet enabler permet aux utilisateurs finaux de gérer leurs listes de contacts et les informations de profil de plusieurs applications en utilisant différents accès. Les informations sont stockées dans le réseau et elles peuvent être accédées ou modifiées à partir d'une application client si elle implémente un client XDM.

2.2 La gestion des entrées sorties dans l'ASR

L'ASR a principalement un rôle de proxy dans le signalement et la communication et il doit traiter un nombre très important de messages. Ce traitement induit parfois de longues pauses pour attendre les réponses de composants voisins. La seule manière d'optimiser l'occupation de la plateforme est de traiter les tâches massivement en parallèle et de manière asynchrone.

2.2.1 Le pattern Reactor et Proactor et l'ASR

2.2.1.1 Le principe des patterns Reactor et Proactor

La concurrence ne peut être traitée uniquement par des threads car ils consomment trop de ressource mémoire. Ils sont souvent utilisés uniquement pour attendre une réponse, c'est-à-dire qu'ils tournent à vide. La solution d'Alcatel-Lucent est d'utiliser les patterns nommés Reactor et Proactor pour implémenter des entrées sorties diminuant la redondance [4] et supportant la mise à l'échelle.

Le pattern Reactor [28] permet de démultiplexer et distribuer les requêtes de service d'un ou de plusieurs clients vers une ou des applications. Ce principe applique le principe d'Hollywood qui est "ne nous appelle pas, nous t'appellerons".[27] Les applications sont passives, car elles ne sont responsables que du traitement des événements. Elles attendent que le réacteur leur envoie des requêtes. Pour être général, nous utiliserons le terme d'handler d'événements à la place d'applications.

Le réacteur a la responsabilité d'attendre des événements pour les distribuer à des handlers d'événements en fonction du type de travail. Le pattern Reactor souffre de quelques contraintes. Il ne supporte pas bien la mise à l'échelle surtout pour des appels simultanés ou de longues durées, car les arrivées de tous les appels sont gérées et démutiplexées au même endroit et les appels sont traités de manière synchrone.

Le pattern Reactor a été amélioré pour donner le pattern Proactor.[27, 25] Il permet la gestion des entrées sorties de manière asynchrone grâce à un mécanisme de trigger sur la complétion d'une opération asynchrone. [5] Quand ces opérations sont complétées, un processeur d'opération asynchrone et le proacteur collaborent pour démultiplexer l'événement de complétion [6] et pour l'associer à l'événement déclencheur. La réponse peut alors être retournée vers le client.

2.2.1.2 Les réacteurs dans l'ASR

Soit, l'ASR est entièrement monothreadée. Un réacteur [7] qui distribue tous les événements vers des handlers d'événements (voir section 2.2.2).

4. L'utilisation de threads induit une redondance importante.
5. L'opération asynchrone consiste à traiter un événement par un handler d'événements. Le traitement est asynchrone dans le sens où le handler d'événements rend directement la main au réacteur.
6. L'événement de complétion est le résultat fournit par l'handler d'événements
7. Par abus de langage, on utilise le terme réacteur à la place du terme proacteur. Il s'agit bien du pattern Proactor car les requêtes sont traitées de manière asynchrone.

Soit, l'application est multithreadée par protocole. Les événements sont réceptionnés directement par le réacteur principal qui les redistribue à des réacteurs spécifiques à un protocole ou simplement à un réacteur en charge d'une activité importante. Il existe dans l'ASR le réacteur principal, le réacteur SIP, le réacteur HTTP, le réacteur qui gère toutes les transactions distribuées, etc.

Un réacteur est un thread qui attend sans cesse qu'une donnée soit à lire ou à écrire sur un socket, il distribue des tâches aux handlers d'événements de manière asynchrone et réceptionne les événements de complétion.[8] L'attente d'événements est implémentée par la fonction poll() présente dans l'algorithme 1. [7] Les réacteurs communiquent entre eux via des sockets.

Algorithme 1 La boucle du réacteur

```
1: while true do
2:     poll()
3:     if readable then read()
4:     dorequest()
5:     if writable then write()
6:     do something
7: end while
```

2.2.1.3 L'utilisation du pattern Proactor dans l'ASR

La figure 2.5 montre une vue haut niveau du pattern Proactor implémenté dans l'ASR d'Alcatel-Lucent. Une requête est envoyée par un client et arrive dans l'agent. Le réacteur détecte l'arrivée de cet événement grâce à la fonction poll() et lit le contenu de son socket pour invoquer la méthode doRequest() du handler d'événements capable de traiter cette requête. Cet appel est asynchrone, la main est immédiatement rendue au réacteur principal qui continue sa boucle (voir l'algorithme 1).

Lorsque la requête est traitée, le résultat est placé dans une file de résultats du réacteur à destination du socket d'écriture. Il est prévenu grâce à l'écriture d'un bit dans le socket du réacteur principal, cette écriture représente l'événement de complétion. Le réacteur détecte la demande d'écriture grâce à la fonction poll() et exécute sa boucle pour retourner le résultat au client correspondant à la requête. On devine bien que l'ASR est manifestement basé intensivement sur l'utilisation de files d'attente. Il existe les files pour stocker les événements (les requêtes) et les événements de complétion (les résultats). Ce genre de files sont disponibles pour chaque réacteur et sont monitorées par le metering service (voir la section 2.4).[9]

2.2.2 Le thread pool

L'utilisation du pattern Proactor nécessite de développer des API asynchrones. Elles implémentent des handlers d'événements que le réacteur leur distribue. Toutes les tâches ne sont pas toujours gérées par des API asynchrones, car elles ne sont pas encore implémentées. Il est alors indispensable d'avoir recours à des threads. Il est impératif de passer par le gestionnaire de thread pour en lancer un, sinon la plateforme perd les avantages liés à l'application du pattern Proactor. Le gestionnaire est nommé thread pool et est représenté sur la figure 2.6.

2.2.3 Les transactions distribuées de sessions

Les transactions offrent les propriétés ACID qui rendent la conception des applications plus simples et robustes. Ce service est offert par une API asynchrone.[10] Elle permet de gérer les transactions à réaliser sur des sessions. [5, 29] Les sessions distribuées sont accessibles à partir de n'importe quel agent dans un groupe. Une fois une session créée, n'importe quel agent peut accéder ou changer la valeur des attributs de la session ou même la supprimer.

8. Il a également la possibilité d'exécuter des tâches plus tard dans le temps avec des Timers (voir section 2.2.4)

9. Toutes les informations données dans la section Reactor et Proactor ne correspondent pas aux détails d'implémentation des entrées sorties de la plateforme d'Alcatel-Lucent. Les détails d'implémentation me sont inconnus, mais seul le principe est important. Il faut retenir que l'ASR fonctionne beaucoup sur le principe de files d'attente.

10. C'est toujours une implémentation du handler d'événements au sens pattern Reactor/Proactor.

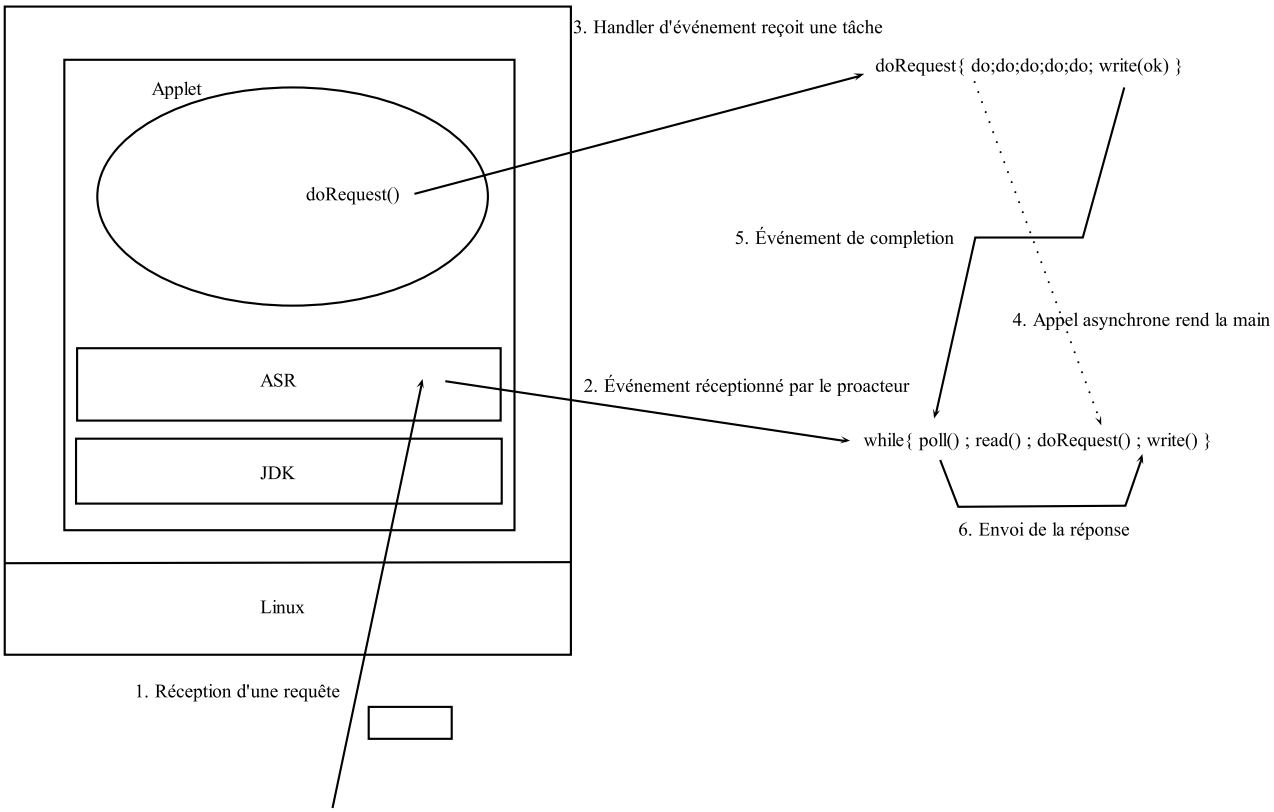


Figure 2.5 – Le pattern Proactor est implémenté dans l'ASR.

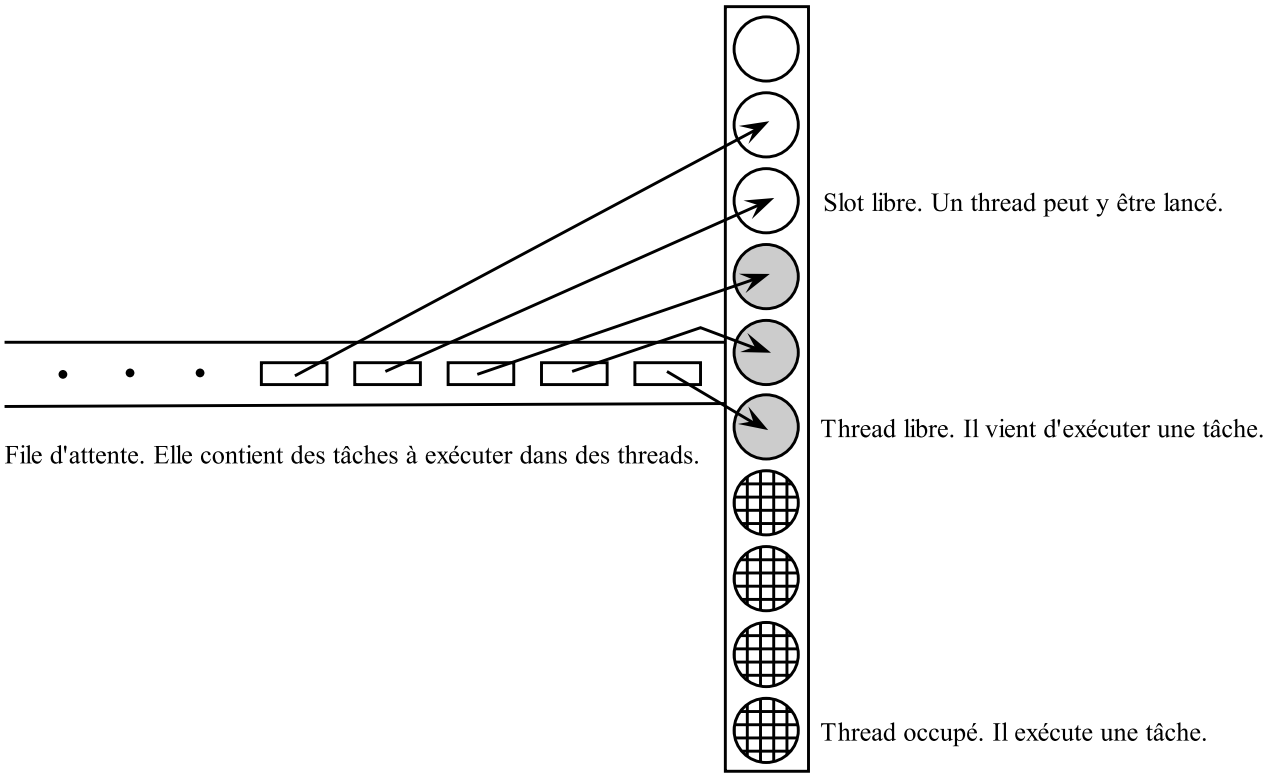


Figure 2.6 – Le thread pool.

Le stockage d'information dans des sessions permet aux applications de gérer les erreurs du contrôle de partage de la charge. Des messages subséquents pourraient être par erreur envoyés à un mauvais agent. Celui-ci peut continuer le traitement initié par l'autre agent en accédant aux données grâce à la distribution des sessions.

L'ASR est hautement disponible, car la disparition d'un agent active les sessions dupliquées sur l'agent esclave (voir section 2.0.1.1). Le mécanisme des timers présenté dans la section 2.2.4 peut aussi être hautement disponible grâce à l'utilisation du mécanisme d'expiration de session. L'application est notifiée de l'expiration de la session.

2.2.4 Les timers

C'est un mécanisme qui permet répartir ou de programmer l'exécution de tâches dans le temps. Cette file de tâches est parcourue périodiquement par le réacteur principal pour vérifier s'il doit distribuer une tâche à un handler d'événements. Ce travail est réalisé dans la partie doSomething() de l'algorithme du réacteur (voir l'algorithme 1).

L'utilisation du mécanisme des timers est une difficulté supplémentaire que le contrôle d'admission ne peut négliger. Des actions peuvent être différées dans le temps induisant une évolution de la charge d'un agent indépendamment du trafic entrant.

Les timers sont souvent utilisés pour gérer le mécanisme de retransmission de messages après le dépassement d'un timeout. Lorsqu'un timer arrive à échéance, alors l'action de retransmission du message est répétée.

2.3 L'administration de l'ASR

L'administration de la plateforme est centralisée à l'aide d'un autre composant proxy appelé FastCache. Toute la configuration de la plateforme et les informations de suivi [11] y sont stockées.

L'ASR offre plusieurs points d'accès pour administrer la plateforme. Une interface web (avec Tomcat), une interface en ligne de commande (script) et enfin une interface JMX. L'interface JMX nécessite d'utiliser le composant proxy JMXGateway.

Comme pour l'administration, les logs sont centralisés dans la plateforme. Grâce au composant proxy Syslog NG, tout ce qui se passe peut être consigné dans des logs selon le niveau de log choisi. Chaque composant proxy à son fichier de log dédié. La figure 2.7 montre comment la plateforme est monitorée.

Toutes les informations disponibles dans les logs sont organisées de manière hiérarchique et sont définies selon la convention de Log4J. Log4J autorise cinq niveaux de log, le niveau DEBUG, INFO, WARN, ERROR et FATAL, la définition d'un logger est une affectation à un de ces niveaux. Chaque domaine des composants proxy est scindé en domaines plus spécialisés. Comme les loggers sont organisés hiérarchiquement, un logger non défini hérite du niveau de log dans son parent le plus proche. La racine de l'arbre des loggers, le *rootLogger* doit toujours être définie.

Les erreurs fatales, les connexions d'utilisateur, la progression des exécutions, etc., toutes ces informations ne sont pas toujours utiles. En production, seules les erreurs fatales sont enregistrées tandis qu'en développement les logs sont utilisés comme des outils de débogage. Par défaut, le niveau de log est à WARN dans l'ASR, c'est-à-dire que la génération des logs comprendra aussi les informations sur les erreurs et les erreurs fatales.

Les logs générés par les agents possèdent toujours le même format. Il est décomposé en deux grandes parties, un en-tête standard et un message spécifique du composant proxy. L'en-tête standard comprend la date et l'heure de sa création, le nom du thread qui a consigné l'information, le niveau de log et le nom du logger.

La figure 2.8 présente un exemple de définition des loggers du metering service présenté dans la section 2.4. La première ligne active toutes les informations statistiques de la plateforme, tandis que les autres lignes en commentaires permettent de sélectionner l'enregistrement d'une partie des informations statistiques disponibles.

11. Les informations de monitoring

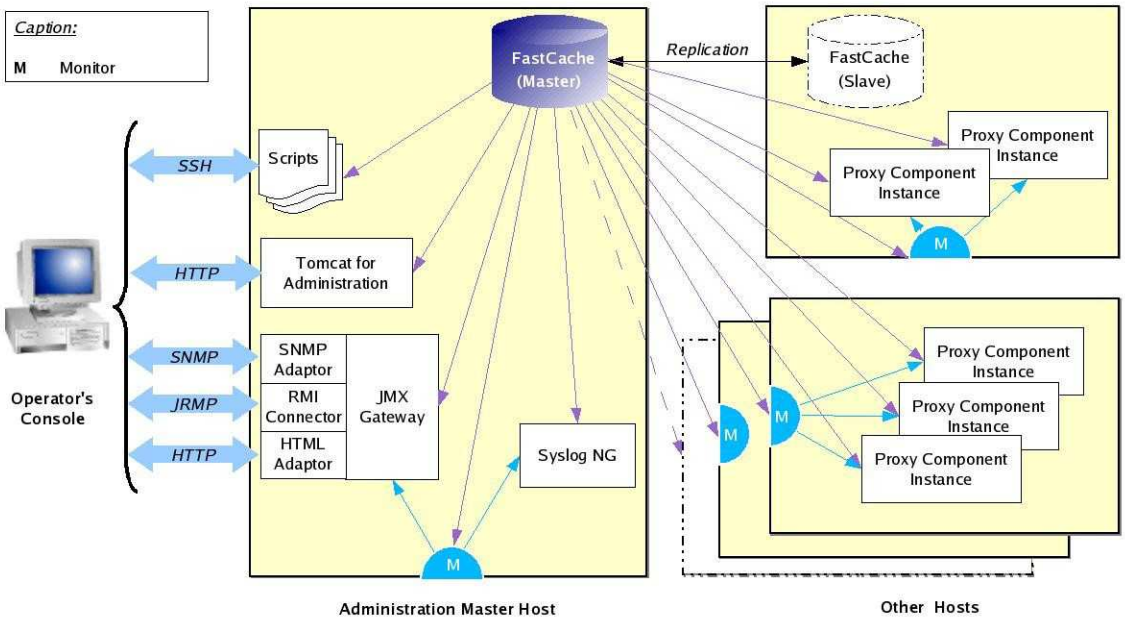


Figure 2.7 – Le schéma global de l'administration de l'ASR.

```
log4j.logger.as.stat=DEBUG
#log4j.logger.as.stat.tpool=INFO
#log4j.logger.as.stat.ds.transaction=INFO
#log4j.logger.as.stat.rtt=INFO
#log4j.logger.as.stat.cpu=INFO
#log4j.logger.as.stat.memory=INFO
```

Figure 2.8 – Un exemple de définition de loggers du metering service.

2.4 Le metering service

Ce service calcule des statistiques (moyenne, écart type, etc.) [6]. Les statistiques sont d'une part accumulées depuis le début de l'exécution de l'ASR et d'autre part réactualisées toutes les 10 secondes. Ce temps est paramétrable et les informations statistiques sont placées dans les logs de l'ASR à condition que ceux-ci soient définis (voir la section 2.3).

Le metering service offre une API qui permet d'instrumenter le code sans en affecter les performances. Il devient possible d'observer le comportement d'un morceau de code dans des conditions de forte charge applicative. [12] Les files du réacteur, le thread pool, les transactions distribuées, etc. peuvent être observés et un outil basé sur gnuplot a spécialement été conçu pour suivre graphiquement l'évolution de ces informations en temps réel. Il est présenté dans l'annexe dans la section 8.3.

Les informations statistiques sont enregistrées toutes les dix secondes dans les fichiers logs dont l'administration est présentée dans la section 2.3. Les annexes reprennent une liste non exhaustive des informations monitorées du réacteur, du thread pool et de l'hôte ainsi que des exemples. (voir la section 8.2).

2.5 Le déploiement standard de l'ASR et des exemples d'utilisation

La figure 2.9 présente un exemple d'un déploiement typique de l'ASR. L'hôte situé en tant que point d'entrée du réseau exécute des io_handlers et des composants d'administration. Si celui-ci disparaît, une station de secours (esclave) prend le relais. Des composants MySql, FastCache et CalloutAgent sont distribués sur les autres hôtes.

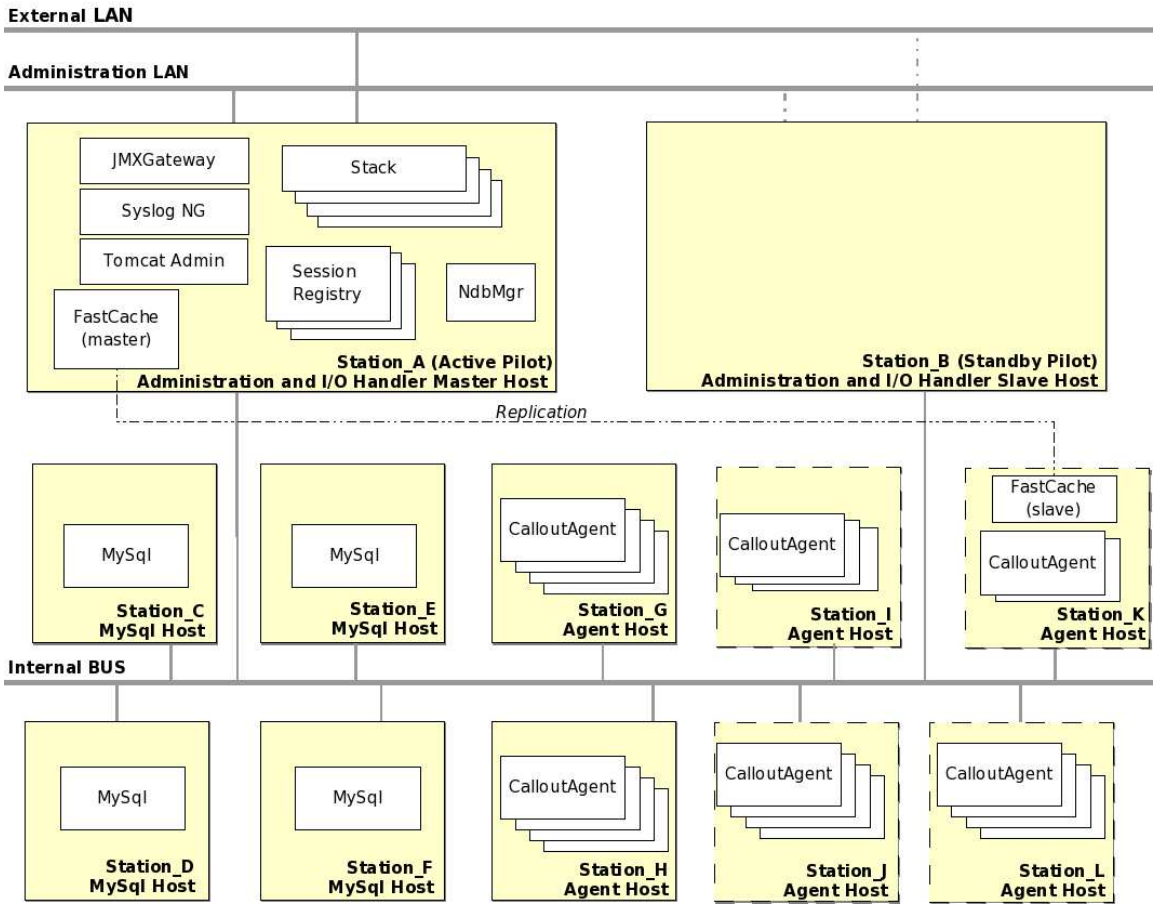


Figure 2.9 – Le déploiement standard de l'ASR.

L'ASR peut être utilisé comme un SIP proxy (figure 2.10) en déployant deux composants proxy, un io_handler SIP

12. On dit qu'on place une sonde

et un agent SIP. L'agent inclut un conteneur SIP compatible avec l'API des SIP Servlet pour gérer le protocole SIP. L'io_handler intercepte le trafic et le redirige vers l'agent qui peut exécuter des actions dessus.

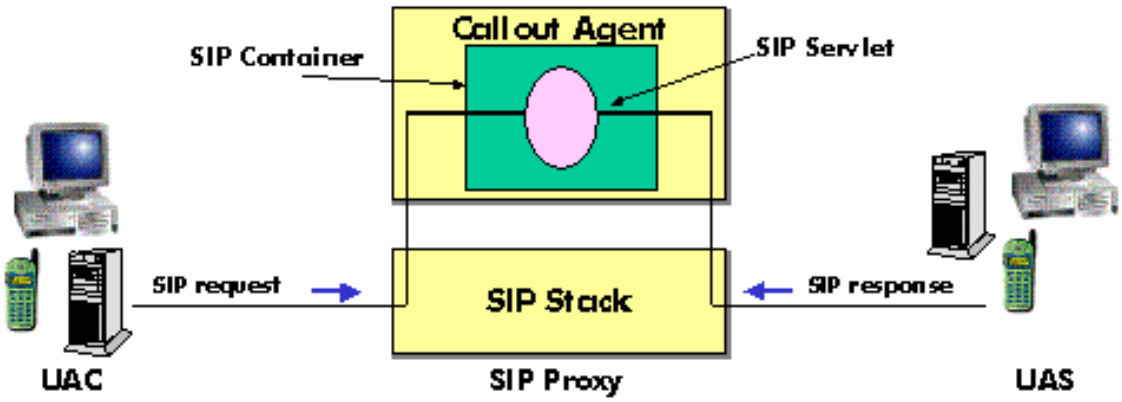


Figure 2.10 – L'ASR est utilisé comme un proxy SIP.

L'ASR est par exemple utilisé comme *user agent client* (UAC) avec un *user agent server* (UAS) (figure 2.11).

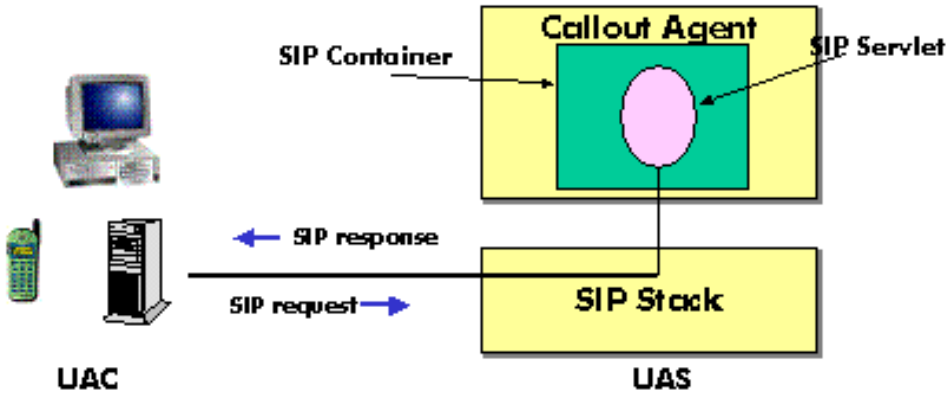


Figure 2.11 – L'ASR est utilisé comme un serveur SIP.

2.6 Conclusion

L'ASR est dotée d'une architecture basée sur le principe de la redondance des composants la rendant particulièrement fiable. La gestion des flux de trafic entre ces composants est assurée grâce à des mécanismes, dont le contrôle du partage de la charge et le contrôle d'admission.

L'ASR est maintenable et gère la complexité grâce à sa découpe en conteneurs et enablers. La ligne de conduite de l'équipe de développement d'Alcatel-Lucent est de développer des fonctionnalités de la plateforme les plus génériques possible, le contrôle d'admission et le partage de la charge en font partie.

L'infrastructure d'administration et de suivi de l'ASR est particulièrement bien développée, spécialement grâce au metering service. Il est possible de suivre et d'observer presque tous les chemins d'exécution de la plateforme. Ces données sont très utilisées et utiles pour comprendre le comportement de l'ASR lors d'une surcharge et leur exploitation par le contrôle d'admission ne laisse aucun doute.

Chapitre 3

La stratégie de contrôle mise en place et les objectifs des algorithmes de contrôle

Ce chapitre présente les objectifs du contrôle d'admission et de charge ainsi que la stratégie mise en place pour les atteindre. Les algorithmes et leur implémentation sont exposés dans le chapitre 4.

3.1 Les objectifs du contrôle d'admission

Une situation de surcharge doit être rapidement détectée par le contrôle d'admission. Il anticipe et prévient ces situations en respectant à tout moment quatre critères.

1. Le système accepte un maximum de requêtes et refuse les autres s'il ne peut les traiter. Le temps de traitement d'une requête est minimisé, c'est-à-dire que les temps d'attente doivent être minimaux.
2. Si le système a atteint sa capacité nominale et si le taux de trafic injecté augmente, alors le taux de trafic accepté par le système reste constant dans le temps. Dans cette condition, l'évolution du taux de trafic refusé et du trafic injecté est identique (voir figure 3.1).
3. Si un client ne reçoit pas de réponse concernant une de ses requêtes après 500 ms (pour SIP), alors il retransmet sa requête uniquement si le protocole de transport est de type *best effort* comme UDP [20]. Le système évite le mécanisme de la retransmission de requêtes en minimisant le temps de réponse. [1]
4. Le système évite le dépassement de délais critiques menant à des situations comme le suicide d'agents ou la perte de connexions.

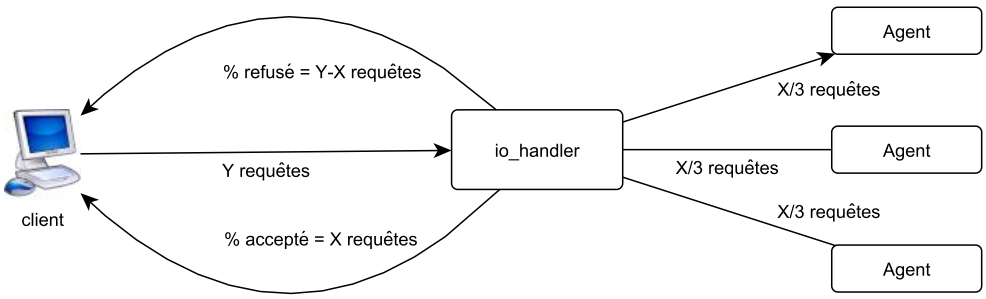


Figure 3.1 – Un io_handler refuse des requêtes si tout ses agents sont surchargés.

1. L'ASR n'a aucun contrôle sur le client.

3.2 Les objectifs du contrôle de charge

Le contrôle de charge doit être impartial car il garantit la cohérence du traitement des messages appartenant à une même session (les messages subséquents). Ils doivent tous être envoyés au même agent. Les sessions ont une définition large car elles sont soit des cookies pour le protocole HTTP, soit des dialogues SIP pour le protocole SIP.

En résumé, les objectifs du contrôle de charge sont :

1. Les agents partagent la même quantité de nouveaux messages.
2. Les messages subséquents sont toujours envoyés au même agent.

3.3 La stratégie de contrôle mise en place pour atteindre les objectifs

Un io_handler possède une importante mission en tant que point d'entrée de la plateforme d'Alcatel-Lucent (voir section 2.0.1). Il évalue d'une part la charge du système et d'autre part il prend la décision critique de refuser des messages.

L'idée générale de l'algorithme de contrôle d'admission est d'utiliser des tokens pour autoriser l'envoi de messages vers les agents sans surcharger le système (voir la figure 3.2). Une quantité W, variable et paramétrable, de messages est transmise tant que l'io_handler possède son token. Les messages sont refusés tant que le token n'est pas revenu à l'io_handler. Il y a autant de tokens qu'il y a d'agents dans un groupe, car chaque token est spécifique à un agent.

Les messages, représentés par des triangles, sont traités ou consommés par les agents, tandis que les tokens, représentés par des ronds noirs, y sont retenus. Ce délai de rétention (RTT_agent secondes) équivaut à la durée totale du traitement de tous les messages précédant le token au moment de son arrivée dans l'agent. Le délai RTT mesuré par l'io_handler entre l'émission et la réception de son token lui permet de mesurer la charge globale du système afin d'ajuster la transmission de messages.

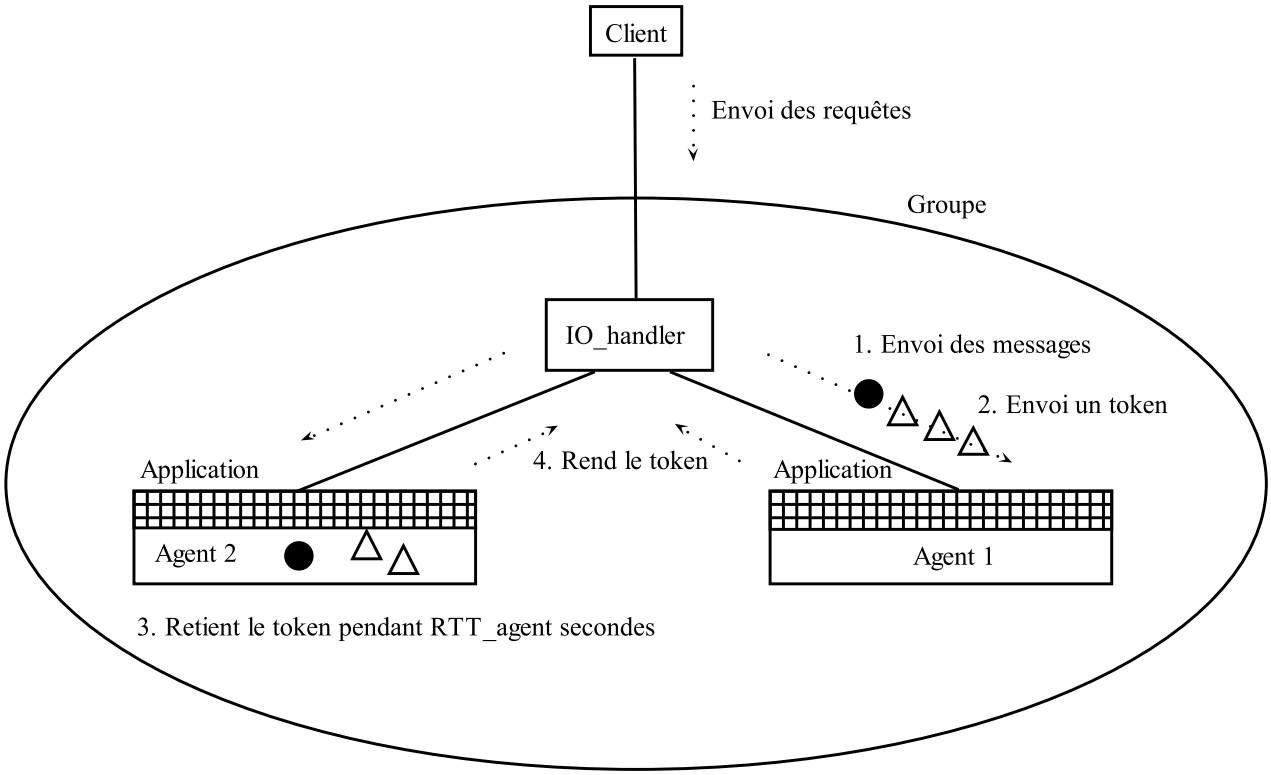


Figure 3.2 – Le principe de l'algorithme de contrôle d'admission dans l'ASR.

La modulation de l'envoi de messages est réalisée par un algorithme à fenêtre. Son idée principale est de permettre à un proxy de transmettre un certain nombre de messages avant de recevoir une confirmation de leur réception ou de leur

traitement (figure 3.3). Chaque émetteur maintient une fenêtre qui permet de limiter le nombre de messages transitant dans le réseau sans avoir été confirmés.

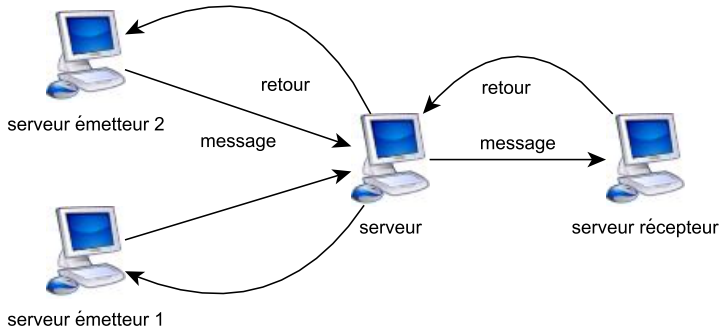


Figure 3.3 – L'illustration du concept de serveur émetteur et de serveur récepteur.

Chaque émetteur possède un compteur de messages non confirmés pour chacun de ses serveurs récepteurs. Chaque fois qu'une confirmation est reçue alors le compteur est décrémenté. L'émetteur arrête d'envoyer du trafic à un serveur récepteur lorsque le compteur associé de messages non confirmés dépasse la longueur de la fenêtre.

Une fenêtre possède une longueur initiale et est réactualisée à chaque réception de confirmation de messages. La fenêtre grandit ou diminue en fonction du retour de son récepteur. Il est courant de considérer qu'il est nécessaire de diminuer la longueur de la fenêtre si le RTT est plus grand qu'un seuil et de l'augmenter dans le cas contraire. Si la longueur d'une fenêtre devient plus petite que la valeur du compteur du nombre de messages non confirmés, alors l'émetteur attend de recevoir des confirmations pour continuer la transmission des messages. Si la longueur de la fenêtre devient nulle, alors l'émetteur doit continuer à recevoir des retours de son récepteur s'il veut un jour à nouveau incrémenter sa fenêtre pour envoyer du trafic [30].

La longueur d'une fenêtre est un paramètre crucial car elle détermine le taux de messages transmis. Une bonne fenêtre doit trouver un compromis entre le taux de trafic transmis et le temps de traitement des messages car, si la longueur de la fenêtre augmente, alors le taux de messages transmis augmente, mais cela amène les messages à passer plus de temps dans les files d'attente [31] (et par conséquent, le RTT du token augemente et contribue à la diminution du taux de messages transmis).

Le token est censé revenir lorsque tous les messages le précédant sont traités, mais cette vision est idéaliste et elle n'est pas réalisable dans l'ASR (voir la section 3.4.2). Le contrôle d'admission est situé dans une couche basse de l'ASR, il est multiprotocole et générique, dès lors, la seule possibilité de connaître les temps de traitement des messages serait que la couche applicative les fournisse, mais ce n'est pas la stratégie choisie par Alcatel-Lucent. Pour contrer cette limite, Alcatel-Lucent utilise le RTT comme un indicateur de charge d'un agent : plus il est élevé et plus l'agent est chargé.

La notion de RTT, analysée à travers le traitement des messages (voir la section 3.4.2), justifie l'application d'une stratégie statistique d'évaluation de la charge d'un agent. Un développement détaillé est présenté dans la section 4.

3.4 La notion de RTT

Cette section présente la notion de RTT et la combine avec la notion de traitement des messages.

3.4.1 Une définition du RTT

Le déplacement des données entre les entités d'un réseau prend du temps. Les sections suivantes proposent différentes définitions du RTT qui est le temps moyen de voyage d'un signal entre deux points dans un réseau.

3.4.1.1 Le RTT en toute simplicité

Le RTT est composé de différents types de délais. Les délais de routage d'un paquet à chaque routeur ($d_{routage}$), les délais de mises en file d'attente ($d_{attente}$), les délais de transmission de la donnée (d_{trs}) et les délais de propagation des paquets entre les routeurs (d_{propag}). [22]

$$d_{noeud \rightarrow noeud} = N(d_{routage} + d_{attente} + d_{trs} + d_{propag}) \tag{3.1}$$

L'équation 3.1 représente le temps nécessaire à une donnée pour voyager entre deux noeuds séparés par N-1 routeurs (hypothèse de réseau parfaitement homogène). L'équation 3.2 représente le RTT entre deux noeuds d'un réseau. Le délai RTT_agent est ajouté au temps de voyage d'une information entre deux noeuds distants, il représente le temps nécessaire pour annuler la charge de l'agent (c'est le temps nécessaire pour lui transmettre à nouveau des messages à traiter).

$$RTT = 2d_{noeud \rightarrow noeud} + RTT\_agent \tag{3.2}$$

3.4.1.2 Le RTT dans une plateforme FIFO

Le RTT entre un io_handler et un agent correspond au voyage d'aller-retour d'un token. Si on considère que des messages sont envoyés à un agent à un taux donné et **que chaque message est traité en une seule opération**, alors le RTT correspond à la somme des temps de traitement de tous les messages non consommés précédant le token au moment où il entre dans l'agent, à laquelle s'ajoute les autres délais représentés dans l'équation 3.1. Ce comportement est suggéré dans la présentation des algorithmes à fenêtre où l'envoi de chaque message ou d'un ensemble de messages est confirmé (On envoie un token tous les W messages).

L'équation 3.3 représente le RTT entre un io_handler et un agent si ce dernier est monothreadé et possède un taux de service de μ messages par seconde. L'io_handler implémente un algorithme à fenêtre dont la longueur vaut W, qui permet de limiter le trafic au taux de λ messages par seconde.

$$RTT = \max\left(W(\frac{1}{\mu} - \frac{1}{\lambda}), 0\right) + 2d_{noeud \rightarrow noeud} \tag{3.3}$$

Si le taux de service μ est négligeable par rapport au taux de messages λ ou si λ vaut l'infini, alors l'équation 3.3 se simplifie en l'équation 3.4.

$$RTT = W(\frac{1}{\mu}) + 2d_{noeud \rightarrow noeud} \tag{3.4}$$

Si l'agent est multithreadé et possède n threads, alors on réécrit l'équation 3.4 en 3.6 et l'équation 3.3 en l'équation 3.5.

$$RTT = \max\left(\frac{W}{n}\frac{1}{\mu} - W\frac{1}{\lambda}, 0\right) + 2d_{noeud \rightarrow noeud} \tag{3.5}$$

$$RTT = \frac{W}{n}\frac{1}{\mu} + 2d_{noeud \rightarrow noeud} \tag{3.6}$$

Les modèles de Dimitri Tombroff et de Michaël Bouhy considèrent l'ASR comme une plateforme FIFO multithreadée au chapitre 5.

3.4.1.3 Le RTT dans l'ASR

L'équation 3.7 présente le RTT entre l'io_handler et un agent et se base sur l'équation 3.5, mais le terme représentant la durée de traitement des messages présents dans le système quand le token arrive est remplacé par une fonction F dépendant de la longueur W de la fenêtre et de λ.

$$RTT = F(W,\lambda) + 2d_{noeud \rightarrow noeud} \tag{3.7}$$

Le traitement d'un message est décomposé en N opérations unitaires dont A s'exécutent en séquence (pendant opSeq secondes) et B en parallèle (pendant opPara secondes) (équation 3.8).

$$N = A + B \tag{3.8}$$

$$F(W,\lambda) = \max\left(W\left(\sum_{i=0}^{A} opSeq(W,\lambda)_i + x \in \mathbb{R} : \forall j \in \mathbb{N}, 0 \leq j \leq B, x = \max(opPara(W,\lambda)_j)\right) - \frac{W}{\lambda}, 0\right) \tag{3.9}$$

Les délais des opérations, opSeq et opPara, peuvent être réexprimés en fonction de W et λ grâce à la théorie des files d'attente. Plus le trafic est important, plus les opérations unitaires prennent du temps à s'exécuter, car les délais d'attente (exprimés en fonction de W et λ) s'ajoutent aux délais de traitement proprement dit.

La fonction F exprime la charge d'un agent en fonction de la quantité de trafic reçu à un taux λ (voir l'équation 3.9). F représente le temps nécessaire à l'agent pour traiter tous les messages au moment où le token arrive. Mais les quantités N, A et B sont inconnues des algorithmes de contrôle rendant l'implémentation de cette équation irréalisable (voir la section 3.4.2). [2]

Les quantités N, A et B étant inconnues, il est nécessaire d'adapter la fonction F de l'équation 3.9 en la fonction F' de l'équation 3.10. L'implémentation présentée au chapitre 4 est fidèle à cette dernière équation. La différence est importante, car le RTT du token ne représente plus la durée de traitement de tous les messages le précédant au moment où il entre dans l'agent. Alcatel-Lucent utilise alors le RTT comme un indicateur de charge d'un agent.

$$F'(W,\lambda) = W\left(\sum_{i=0}^{A} opSeq(W,\lambda)_i\right) + x \in \mathbb{R} : \forall j \in \mathbb{N}, 0 \leq j \leq C, x = OP\_STAT(opPara(W,\lambda)_j) \tag{3.10}$$

OP_STAT représente différents opérateurs statistiques utilisés ou développés pour traiter les durées de toutes les opérations s'exécutant en parallèle dans la plateforme (opPara). La quantité C est le nombre d'opérations nécessaires pour traiter les messages au moment où le token entre dans la plateforme.

Les fonctions F et F' ne dépendent pas du taux de service μ de l'ASR, car c'est une notion trop agrégée. $\frac{1}{\mu}$ est le temps moyen de service, c'est-à-dire qu'il correspond au temps nécessaire pour traiter un message en entier, mais ce temps est inconnu dans l'ASR (sauf par les applications ...). Le traitement des messages est décomposé en opérations élémentaires exécutées en mode séquentielle et/ou parallèle. opSeq représente la durée d'une opération en mode séquentiel et opPara représente la durée d'une opération en mode parallèle ($opSeq \leq \frac{1}{\mu}$ et $opPara \leq \frac{1}{\mu}$). Plus les temps opSeq et opPara sont petits, plus $\frac{1}{\mu}$ est petit. opPara est en réalité un temps de service de mysql, ou de n'importe quel autre composant voisin de l'agent.

La deuxième partie de l'équation 3.10 représente une évolution importante de la boucle de contrôle, car elle permet aux io_handlers de mieux évaluer la charge de l'entièreté du système.

3.4.2 Le traitement des messages

Il est possible d'imaginer le traitement de messages par un agent, car il est toujours composé d'opérations fines et indivisibles parfaitement connues et dépend des choix d'implémentation et des protocoles en vigueur. L'ASR est un système à files et les messages sont traités en opérations plus fines telles que le lancement de threads dans le thread pool ou la réalisation de transactions distribuées (voir la section 2.2).

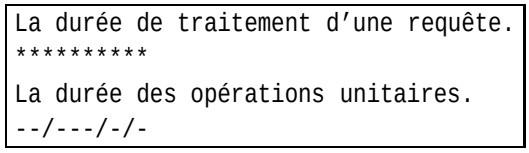
La durée de traitement d'un message est liée au nombre d'opérations unitaires nécessaires à son traitement complet, à la durée de ces opérations ainsi qu'à la concurrence d'exécution de celles-ci. Il est par conséquent impossible pour l'ASR de calculer le temps de traitement d'une requête à partir des temps de traitement de ces opérations, car la réalité est un

2. La réalité est encore plus complexe, car les équations n'expriment pas la variabilité du trafic. Les quantités N, A et B sont évidemment spécifiques à un message.

mélange de plusieurs configurations possibles, impossible à prévoir. De plus, l'algorithme de contrôle d'admission se situe en dessous de la couche protocolaire et applicative, il n'a pas la connaissance de l'enchevêtrement des différentes opérations unitaires. Seule la couche applicative possède cette information.

3.4.2.1 Le traitement séquentiel d'un message

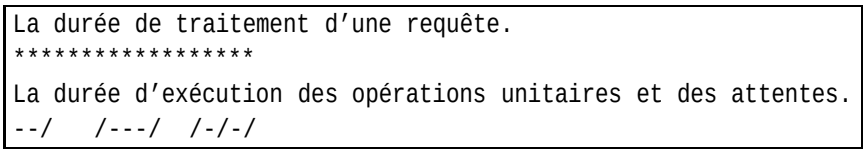
Une étoile $*$ représente une unité de temps utilisée par un agent pour traiter une partie d'un message (par exemple un SUBSCRIBE). Un tiret $-$ représente une unité de temps d'exécution d'une opération unitaire (par exemple une requête SQL). Le slash / sépare l'exécution des opérations et des délais d'attente entre celles-ci. La figure ci-dessous met en relation la durée du traitement d'un message avec la durée d'exécution de chacune de ses opérations unitaires.



Si une requête est traitée uniquement en séquence alors la somme des temps de traitement de toutes ses opérations unitaires détermine son temps de traitement.

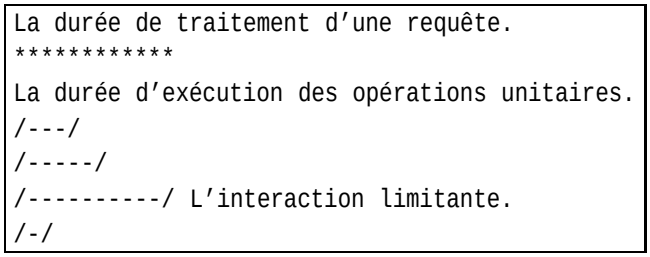
3.4.2.2 Le traitement séquentiel d'un message avec des retards et des temps d'attente

Si une requête est traitée uniquement en séquence, mais avec des retards, la somme des temps de traitement de toutes les opérations unitaires et des temps d'attente détermine le temps de traitement d'une requête.



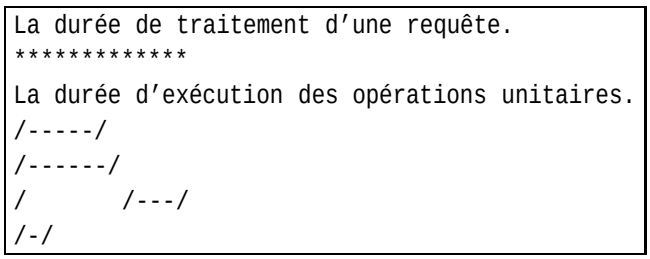
3.4.2.3 Le traitement parallèle d'un message

Les opérations unitaires s'exécutent en parallèle. L'interaction limitante détermine la durée de traitement d'une requête. C'est-à-dire que l'opération la plus longue détermine la durée d'exécution de la requête.



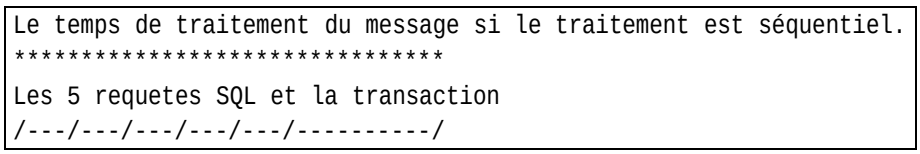
3.4.2.4 Le traitement parallèle d'un message avec des retards et des temps d'attente

Les opérations unitaires s'exécutent avec décalage dans le temps à cause des temps d'attente dans les files d'attente. L'interaction limitante détermine la durée du traitement d'une requête.

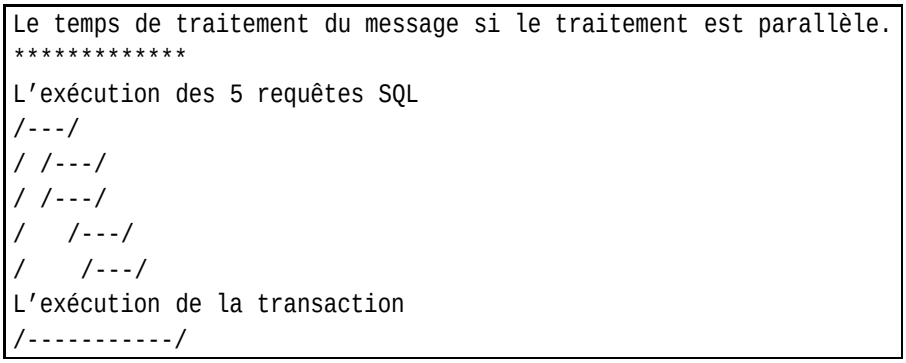


3.4.2.5 Un exemple concret

Les opérations unitaires sont, dans cet exemple, des requêtes SQL et des transactions distribuées. Le traitement d'un message nécessite cinq requêtes SQL et une longue transaction. Les requêtes SQL s'exécutent soit en séquence, soit en parallèle.



Il est possible d'optimiser le traitement d'un message en parallélisant l'exécution des requêtes SQL.



Le traitement du message prend autant de temps que l'exécution de la transaction.

3.5 Conclusion

Le rôle des mécanismes de contrôle du trafic peut se résumer en une seule phrase, il s'agit d'accepter un maximum de messages et de les traiter le plus rapidement possible. Pour atteindre ces objectifs, Alcatel-Lucent a mis en place une stratégie de contrôle d'admission basée sur l'utilisation d'un token et d'un algorithme à fenêtre.

Le RTT du token est mesuré par le point d'entrée du trafic dans la plateforme (l'io_handler) et il le perçoit comme une mesure de la charge globale de celle-ci. Ce mécanisme permet aux composants qui traitent les messages (les agents) de demander aux io_handler de leur transmettre des nouveaux messages, ou au contraire de les refuser.

L'expression mathématique du RTT dépend fortement de la manière dont sont traités les messages dans les agents. Le traitement des messages peut être mono ou multi-threadé, décomposé en opérations plus fines, synchrone ou asynchrone, en séquence ou en parallèle, etc. La durée de ce traintement est inconnue dans l'ASR car trop variable en fonction des messages. Cette limite impose une analyse statistique du comportement de la plateforme pour mesurer un RTT représentant le mieux possible l'évolution de la charge globale du système.

Chapitre 4

L'évolution des algorithmes et de leur implémentation

Ce chapitre présente l'évolution des algorithmes du contrôle d'admission et du partage de la charge au cours du stage. Cette section est basée sur la documentation, le code source d'Alcatel-Lucent et nos expérimentations.[4]

Les algorithmes sont divisés en trois catégories, la première est l'algorithme de partage de la charge, il permet à l'io_handler de répartir équitablement la charge entre ses agents. La deuxième est l'algorithme de contrôle de flux permettant à l'io_handler de refuser ou d'accepter des messages en fonction de la charge du système déterminée par le troisième algorithme d'évaluation de la charge globale du système (Le contrôle d'admission regroupe les algorithmes d'évaluation de la charge et de contrôle de flux).

4.1 L'algorithme du partage de la charge

L'algorithme du partage de la charge est fortement lié à l'algorithme du contrôle de flux et il est représenté à la figure 4.1. L'io_handler tient en permanence à jour une liste d'agents candidats et aptent à prendre en charge un nouveau message. L'io_handler teste leur charge pour détecter les occupés et les retirer de la liste des candidats. Un nouveau message est transmis par l'io_handler vers un des agents candidats choisi au hasard dans la liste pour initier une nouvelle session. Tous les messages sont refusés par l'io_handler lorsque la liste des candidats est vide.

4.2 L'algorithme de contrôle de flux

Cet algorithme d'une grande importance a connu deux versions majeures, la première est la version statique et la deuxième est la version dynamique.

La version statique est basée sur l'analyse des files d'envoi de messages. Le principe est bon, mais est en pratique inefficace, car la surcharge est détectée au moment où elle se produit, il est donc trop tard pour réagir (voir la section 4.2.1). La version dynamique vient corriger ce défaut par l'utilisation d'un système à tokens. Un dialogue est instauré entre l'io_handler et chacun de ses agents pour communiquer des informations de charge globale du système uniquement par le temps de parcours du token entre ceux-ci (voir la section 4.2.2).

4.2.1 La version statique de l'algorithme du contrôle de flux

4.2.1.1 Le principe

Le principe de détection d'une surcharge, par l'io_handler, d'un agent est celui d'un entonnoir délimité en niveaux. le premier niveau est le tampon TCP de réception de l'agent, le deuxième est le tampon TCP d'envoi de messages de

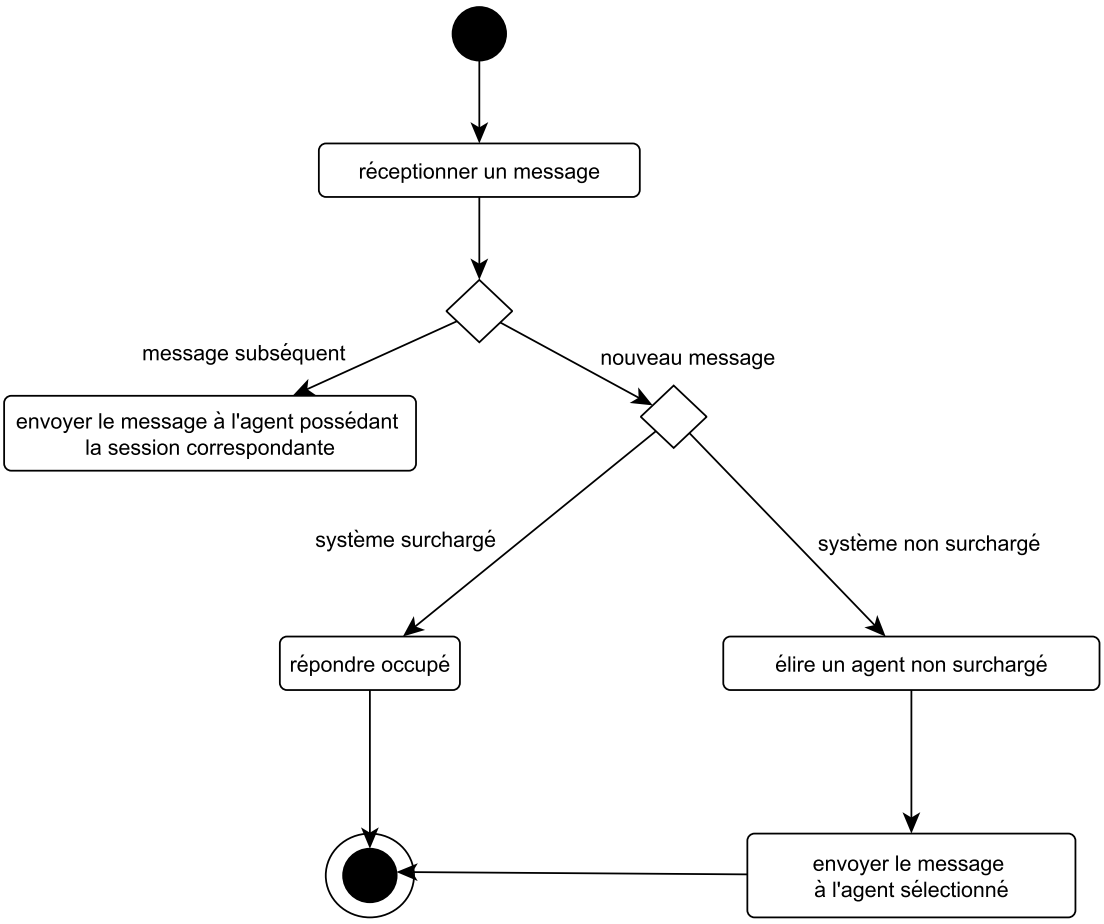


Figure 4.1 – L'algorithme général du contrôle de flux et de partage de la charge. L'io_handler reçoit un message et le transmet à un agent candidat.

l'io_handler et le dernier est la file applicative d'envoi de messages de l'io_handler. Il est seulement autorisé à observer l'état de sa file applicative, c'est-à-dire le dernier niveau de l'entonnoir. Le dépassement d'un seuil de ce niveau, le *lowMuxWaterMark*, déclare l'agent surchargé et permet à l'io_handler de limiter le nombre de messages qu'il lui envoie.

4.2.1.2 La définition de la surcharge du système

La surcharge d'un agent est basée sur l'analyse seule de la file d'envoi des messages de l'io_handler.

```
Si pour tout i, 0<i<=nbrAgent : file applicative d'envoi > lowMuxWaterMark
alors la surcharge du système est détectée.
Si il existe un file applicative d'envoi < lowMuxWaterMark
alors le système est non surchargé.
```

4.2.1.3 La définition de la surcharge d'un agent du système

```
Si il existe un file applicative d'envoi > lowMuxWaterMark
alors l'agent i est surchargé.
```

4.2.1.4 L'algorithme

```
Pour tout message R injecté dans l'io_handler :
        Si R appartient à une session => associée à un agent A
        alors envoyer R à A
        Si système surchargé
        alors répondre BUSY
        Sinon choisir un agent non surchargé A au hasard et envoyer R à A.
```

Il est possible d'activer une option basée sur un deuxième seuil, le *highMuxWaterMark*. Il permet de réinitialiser la connexion entre l'io_handler et l'agent ou de redémarrer l'agent (voir la figure 4.2).

4.2.1.5 Les critiques

La détection de la surcharge d'un agent est réalisée lorsque le remplissage de la file applicative d'envoi dépasse le seuil *lowMuxWaterMark*. Cependant, un agent est réellement surchargé lorsqu'il ne peut plus traiter directement tous les messages qu'il reçoit de la part de son io_handler, c'est-à-dire lorsque ses files de réception de message TCP commencent à se remplir. Malheureusement, l'io_handler ne peut observer que le dernier niveau, c'est-à-dire sa file applicative d'envoi de messages. Lorsque celle-ci commence à se remplir, le tampon TCP de réception de l'agent est plein depuis longtemps. La surcharge d'un agent est détectée tardivement, car il faut attendre que les files d'envoi applicatives prennent le relais. La figure 4.3 montre le cas où un io_handler transmet des messages à un agent avec un taux de transmission du io_handler supérieur au taux de réception de l'agent.

4.2.2 La version dynamique de l'algorithme du contrôle de flux

4.2.2.1 Le principe

La dynamique de cet algorithme vient de l'utilisation, dans l'io_handler, de fenêtres associées à chaque agent. Elles permettent de moduler constamment le nombre de messages envoyés grâce à l'utilisation d'un token. Pour ne pas être surchargé, les agents envoient continuellement leur niveau de charge à l'io_handler.

```
        Si file applicative d'envoi > highMuxWaterMark
        alors réinitialiser la connexion socket MUX et optionnellement
envoyer une requête à l'agent pour qu'il redémarre.
```

Figure 4.2 – Le highMuxWaterMark de la version statique de l'algorithme du contrôle de flux.

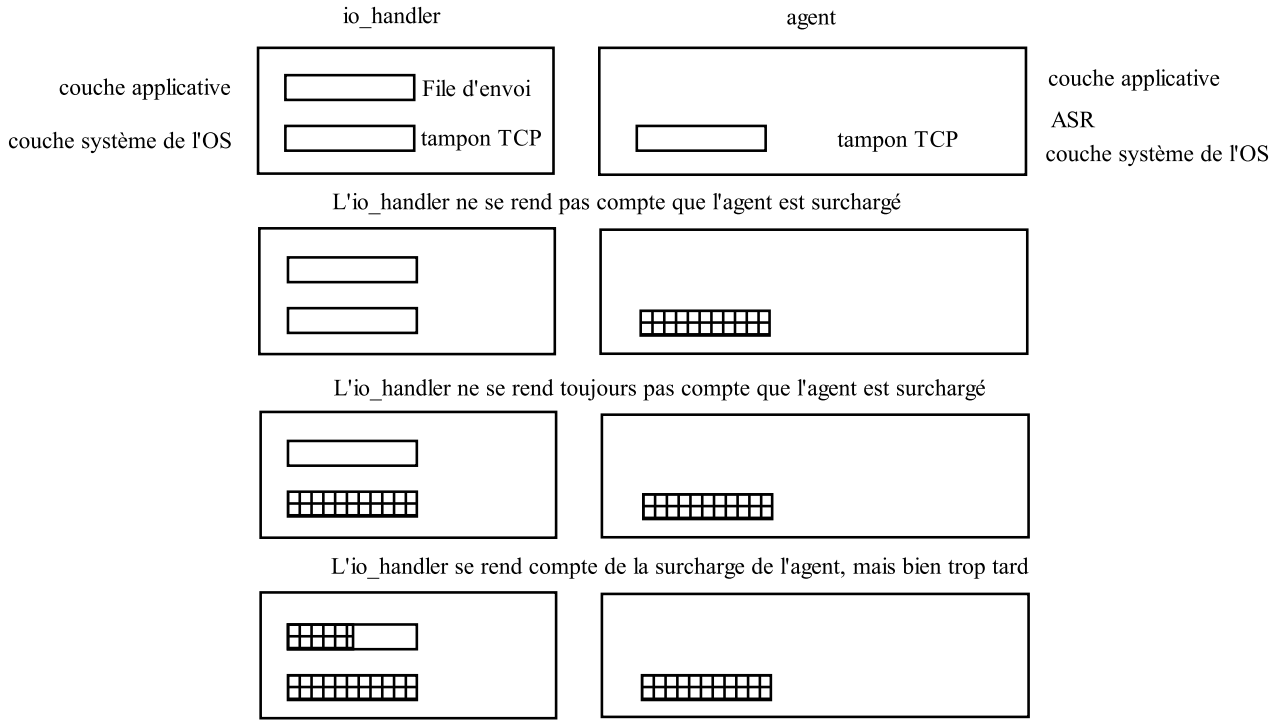


Figure 4.3 – L'illustration du fonctionnement du contrôle de flux statique pendant la surcharge d'un agent.

4.2.2.2 La définition de la surcharge d'un système

La définition de la surcharge du système est entièrement basée sur la longueur des fenêtres. La notion de fenêtre est définie pour la première fois dans la section 3.3 et sa variation de longueur est détaillée dans la suite de ce chapitre.

```
Pour tout i : 0<i<=nbrAgent
        Si fenêtre_i==0
        alors système surchargé
        Si il existe un i : 0<i<=nbrAgent : fenêtre_i > 0
        alors système non surchargé
```

4.2.2.3 La définition de la surcharge d'un agent

La variation de la longueur de la fenêtre suit des règles précises présentées ci-dessous.

```
Si il existe un i : 0<i<=nbrAgent : fenêtre_i==0  OU
Si il existe un i : 0<i<=nbrAgent : fenêtre_i est pleine
de requêtes
        alors agent_i surchargé
Pour tout i : 0<i<=nbrAgent :
        plus la fenêtre_i est petite
        plus l'agent_i est en difficulté.
Pour tout i : 0<i<=nbrAgent :
        plus la fenêtre_i est grande
        plus agent_i est en bonne santé.
```

4.2.2.4 L'algorithme

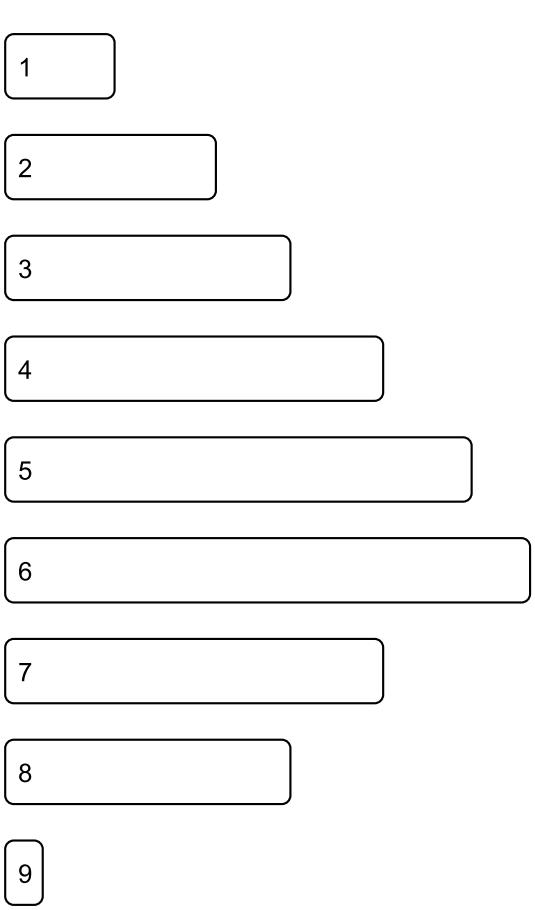
La transmission d'un message de io_handler vers un agent modifie l'état de sa fenêtre. Le taux de nouveaux messages transmis est limité par la longueur de la fenêtre et le RTT du token. [1]

Chaque fois que l'io_handler reçoit le token associé à un agent, sa fenêtre est redimensionnée. Elle est décrémentée si le RTT est plus grand que le seuil autorisé (configurable) ou si le CPU ou la mémoire de l'agent sont trop élevés. Elle est incrémentée si le RTT est petit et si l'utilisation CPU et la mémoire de l'agent sont faibles.

4.2.2.5 Des exemples d'évolution d'une fenêtre

Un exemple de l'évolution de la longueur d'une fenêtre est présenté dans la figure 4.4 et la figure 4.5 illustre l'évolution de l'occupation d'une fenêtre.

1. Attention, le raisonnement est à tenir au niveau logique, il est possible que la couche système segmente les messages pour arriver à tout faire passer dans le réseau.



1. La fenêtre est initialisée à 1

De l'étape 2 à l'étape 6, la longueur de la fenêtre augmente à chaque réception de token. Le RTT de ces tokens est dans chaque cas inférieur au seuil configuré. Par défaut il vaut 50 ms.

6. La fenêtre a atteint sa longueur maximale. Elle reste dans cet état tant que les RTT sont inférieurs au seuil.

7. La réception d'un token dont le RTT est supérieur au seuil provoque la réduction de 25% de sa longueur maximale. Plus la fenêtre est grande, plus vite elle se décrémentera.

8. Idem qu'à l'étape 7.

9. La longueur de la fenêtre est nulle. Elle augmentera uniquement lorsqu'elle recevra un token dont le RTT est inférieur au seuil. L'agent est surchargé.

Figure 4.4 – Un exemple d'évolution de la longueur d'une fenêtre.

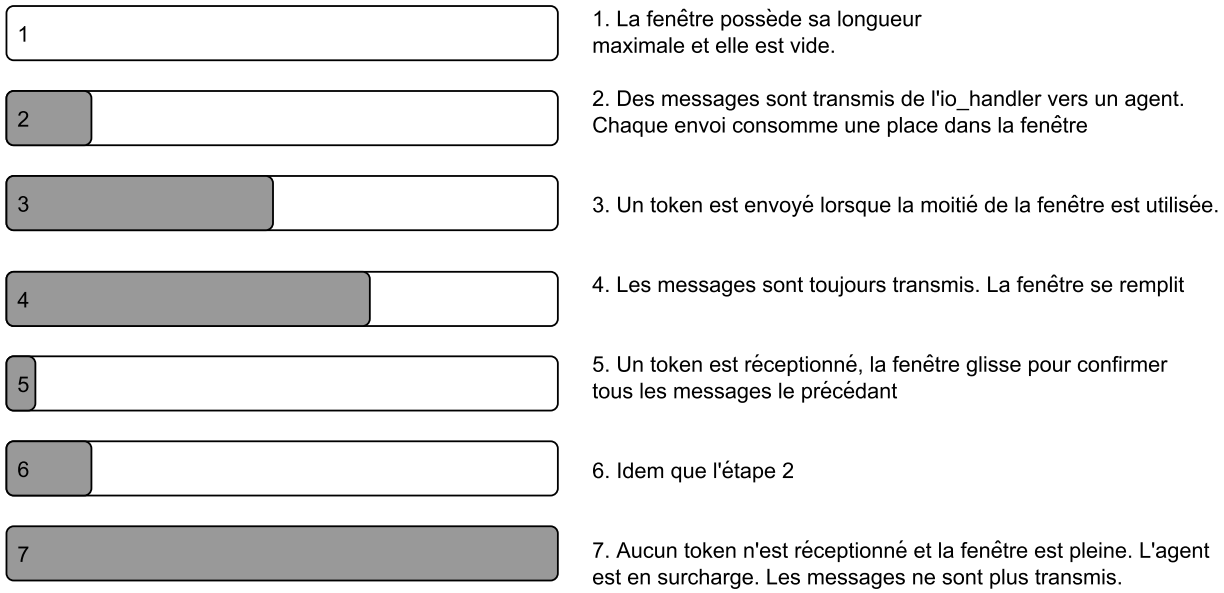


Figure 4.5 – Un exemple d'évolution de l'occupation d'une fenêtre lorsque celle-ci a atteint sa longueur maximale.

Il est possible d'activer une option pour que chaque io_handler se protège en gérant sa propre charge sur base de son CPU. Si l'utilisation des ressources CPU est supérieur à un certain taux (plus ou moins de 90%), alors l'io_handler refuse une partie du trafic injecté pour éviter sa propre surcharge.

4.2.2.6 La critique de la version dynamique

Le principal défaut de cette version dynamique est l'introduction d'un nombre important de paramètres dont le réglage est obscur. Ensuite la version dynamique doit avoir une confiance aveugle dans la mesure de la charge du système qui est réalisée par l'algorithme présenté dans la section 4.3.

4.3 L'algorithme d'évaluation de la charge globale du système

Le contrôle de flux doit se fier au temps de parcours d'un token entre son envoi et sa réception, car lui seul possède l'information de la charge de l'agent et de ses voisins. Si le token donne une information partielle de la charge du système, alors les décisions prises par l'algorithme à fenêtre seront erronées.

Le principe du contrôle d'admission est de retenir un token dans un agent pendant un temps proportionnel à sa charge. L'évaluation de ce temps, le RTT_agent, est très importante et elle est réalisée par un algorithme d'évaluation de la charge globale du système. Nos corrections portent principalement sur cet algorithme et c'est pourquoi cette section présente en détail son évolution.

4.3.1 L'algorithme originel

L'algorithme à token fonctionnait depuis quelques années sans rencontrer de problème. Il traversait l'agent en ne passant que par sa couche MUX (voir section 2.1.1.1). Cette façon de procéder a commencé à montrer ses limites avec des applications communiquant plus fréquemment de manière asynchrone avec d'autres groupes applicatifs, car la charge applicative n'était pas directement mesurée.

Les communications asynchrones n'étaient pas du tout prises en compte dans la mesure du RTT, car le token était directement placé par le réacteur principal dans la file des tâches à écrire sur le socket de sortie vers l'io_handler (voir la

section 2.2.1.3). Les délais de temps "asynchrones" échappaient totalement à cette technique de mesure.

4.3.2 Le token passe par le thread pool

La clé du problème réside dans la capture des temps de communication asynchrone pour détecter une surcharge avant que celle-ci bloque la plateforme ou dégrade fortement ses performances.

L'idée consiste à faire passer le token par le thread pool. Le token arrive dans la couche MUX, le réacteur principal le découvre et il programme son retour vers l'io_handler dans un Runnable, c'est-à-dire que le "token" est placé dans la file d'attente du thread pool. L'exécution du Runnable dans un thread consiste à placer le token dans la file de tâches à écrire vers le socket de sortie.

Cette façon de procéder permet de capter une partie des délais asynchrones, mais d'une manière trop localisée et incomplète. En effet, ce n'est qu'un thread parmi des centaines et son exécution est très rapide. La surcharge n'est détectée que si la file du thread pool devient de plus en plus longue, synonyme de surcharge, sa détection est réalisée trop tard. Les transactions distribuées ne sont pas prises en compte par cette technique de mesure.

Cet essai n'a pas fonctionné, car la surcharge était détectée trop tard, mais a tracé un chemin vers la solution.

4.3.3 L'intervention directe de la connaissance applicative

Prendre en compte la connaissance applicative pour mesurer le RTT était un essai mené parallèlement à l'essai précédent.

L'idée consiste à demander à une application de mesurer un temps moyen de traitements des messages afin de retenir le token dans l'agent pendant cette durée. Le token arrive dans la couche MUX, le réacteur principal le découvre et il programme son retour vers l'io_handler dans un Runnable [2] grâce à un timer. Le réacteur surveille périodiquement la liste des timers pour exécuter ceux qui sont arrivés à expiration. Quand c'est le cas, le token est programmé dans le thread pool. L'exécution du Runnable dans un thread consiste à placer le token dans la file de tâches à écrire vers le socket de sortie.

Cet essai n'a pas fonctionné, car le temps de réaction est beaucoup trop lent. La surcharge est détectée beaucoup trop tard ayant pour conséquence de bloquer la plateforme.

4.3.4 L'utilisation seule de la connaissance de l'agent

Dans un souci de simplicité et de polyvalence de l'algorithme d'évaluation de charge, aucune information n'est demandée à la couche applicative. Tous les délais de communication asynchrone doivent être captés. Ils le sont grâce au placement de sondes dans certaines parties du code de l'agent (metering service, voir le chapitre 2.4). La mesure du RTT se base sur le monitoring des temps de traitement des opérations unitaires (voir la section 3.4.2).

La charge d'un agent est une fonction dépendant des temps issus des différentes interactions avec des composants externes. Seules les interactions asynchrones dans l'agent sont prises en compte. Pourquoi pas les interactions synchrones ? Car une interaction synchrone bloque le système, ce temps est pris en compte lorsque le token traverse l'agent, en effet le token passe par le thread principal (le réacteur). Si celui-ci est bloqué ou ralenti par un traitement plus long, alors le RTT augmente automatiquement. Le token est retenu comme expliqué dans la section 4.3.3.

Parmi les différentes interactions asynchrones, nous en avons identifié deux très importantes, l'activité du thread pool et les transactions distribuées. Les temps d'activités du thread pool permettent d'estimer la surcharge de différents composants externes ainsi que celle des agents voisins. Les transactions distribuées sont utilisées pour la communication interagents, intra ou inter-groupes pour synchroniser différentes sessions. La durée des transactions permet d'estimer la surcharge des agents voisins.

Chaque fois qu'un thread ou une transaction distribuée est exécuté, l'algorithme d'évaluation de la charge reçoit sa durée d'exécution. Une analyse statistique de chaque opération doit être réalisée pour retirer une seule valeur de temps, le RTT_agent, représentant la charge globale de l'agent. Différentes fonctions de traitement statistique de toutes ces données pour mesurer de la manière la plus fiable la charge d'un agent ont été développées et testées, il s'agit de la

moyenne exponentielle, [2] d'une moyenne exponentielle adaptée et d'une fonction spéciale.

4.3.4.1 La moyenne exponentielle et la moyenne exponentielle adaptée

La moyenne exponentielle est une manière simple de traiter les données issues des sondes pour estimer un RTT (voir l'équation 3.10). Elle consiste à pondérer la nouvelle valeur avec la moyenne mobile courante (équation 4.1).

$$\text{moyenne}(i+1) = \alpha * \text{moyenne}(i) + (1-\alpha) * \text{valeur}(i) \tag{4.1}$$

α contrôle la vitesse à laquelle la moyenne s'adapte aux changements, c'est une constante comprise entre zéro et une unité. Plus la valeur de α est petite et plus la moyenne est influencée fortement par les nouvelles valeurs.

Deux fonctions sont testées dans l'ASR, la première est une moyenne mobile avec α=0.7 et la deuxième fonction est une moyenne mobile dont α=0.7 si la valeur est strictement plus grande que la moyenne courante, sinon α=0.95. La figure 4.6 montre que la moyenne mobile adaptée s'adapte rapidement au pic de trafic et diminue selon une exponentielle douce, car une très faible importance est donnée aux nouvelles valeurs.

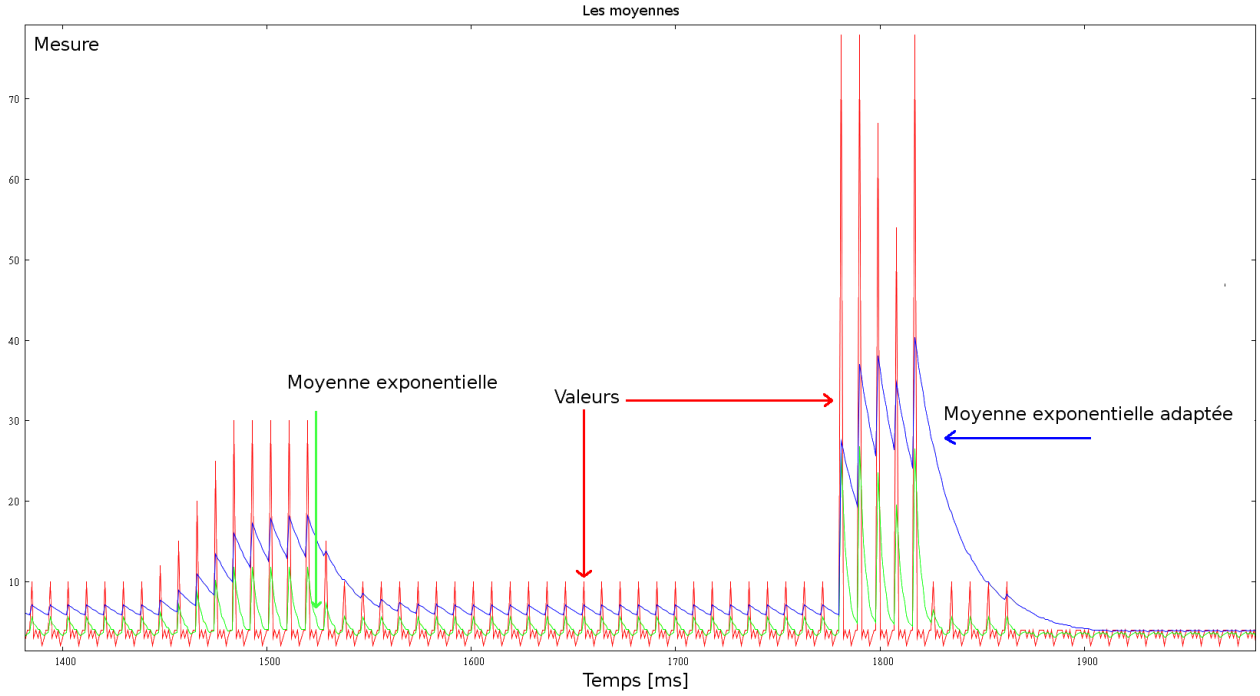


Figure 4.6 – Deux moyennes exponentielles, une avec α=0.7 et l'autre adaptée car α=0.7 si la valeur est strictement plus grande que la moyenne courante, sinon α=0.95.

Si le temps de traitement des messages augmente, alors l'io_handler limite le trafic et il ne connaît cette information que par la durée RTT de voyage du token.

Dans l'ASR, le traitement des messages est composé d'opérations fines et indivisibles (ACID) et il suffit qu'elles prennent plus de temps pour allonger le temps de traitement des messages. Un cas limite, déjà présenté dans la section 3.4.2, est envisagé pour montrer le défaut de cette approche. Il s'agit du traitement parallèle des messages où la durée de l'interaction limitante est peu prise en compte alors qu'elle seule influence le temps de traitement d'un message. L'utilisation de la moyenne exponentielle est déconseillée, car elle risque de noyer (masquer) les durées de l'interaction limitante dans le RTT (voir figure 4.7).

La moyenne mobile adaptée est plus fiable, car elle est plus sensible aux valeurs maximales permettant au RTT représenter plus fidèlement l'interaction limitante (figure 4.8). Elle est plus fiable mais possède le même défaut.

2. On appellera la moyenne mobile à pondération exponentielle, moyenne exponentielle, par abus de langage.

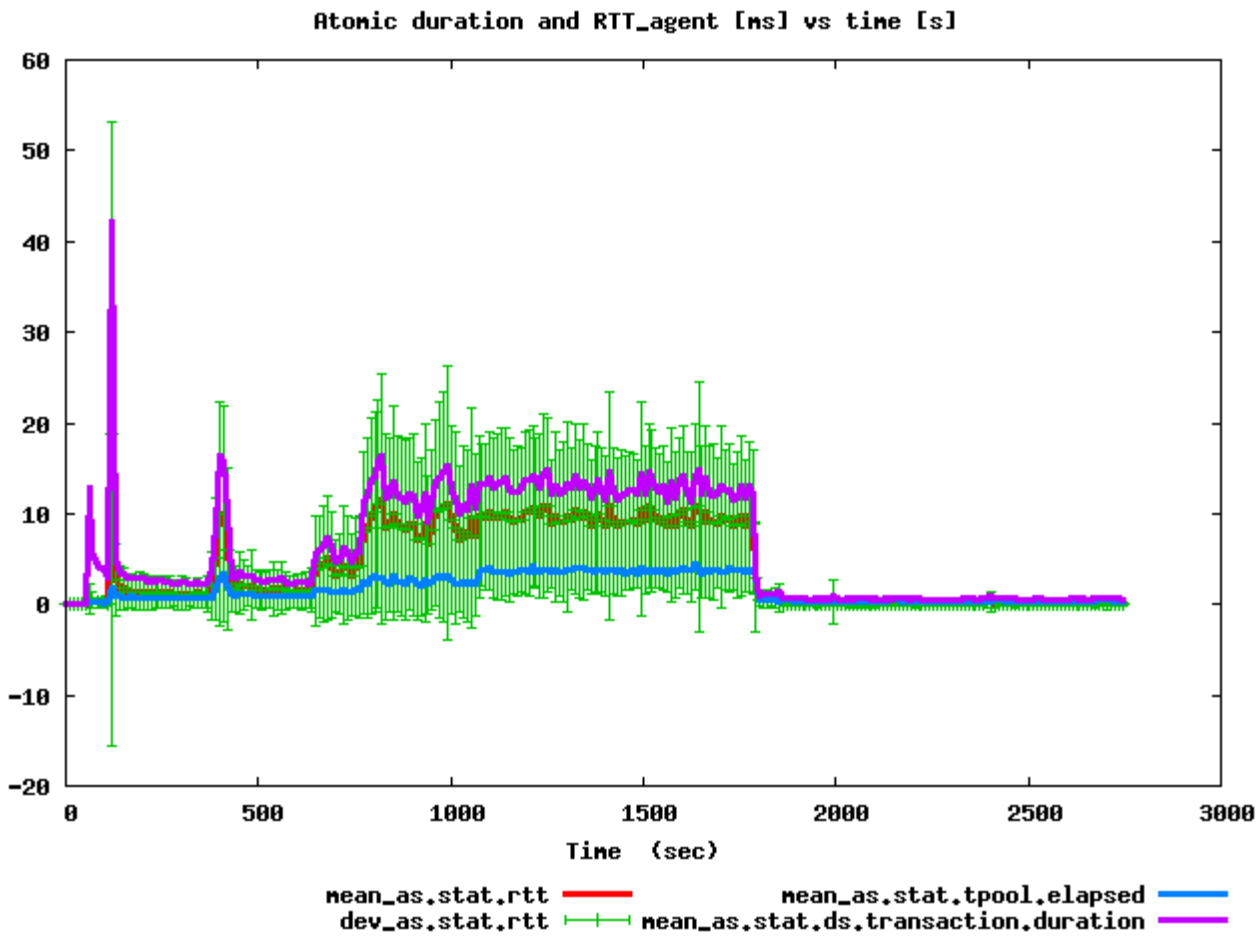


Figure 4.7 – Valeur moyenne des sondes et du RTT calculés grâce à une moyenne exponentielle. Les valeurs statistiques sont réalisées sur des intervalles de dix secondes et la déviation du RTT est représentée par les barres verticales. Les messages sont les SUBSCRIBE GROUP à taux variable présenté dans la section 6.3.2.2.

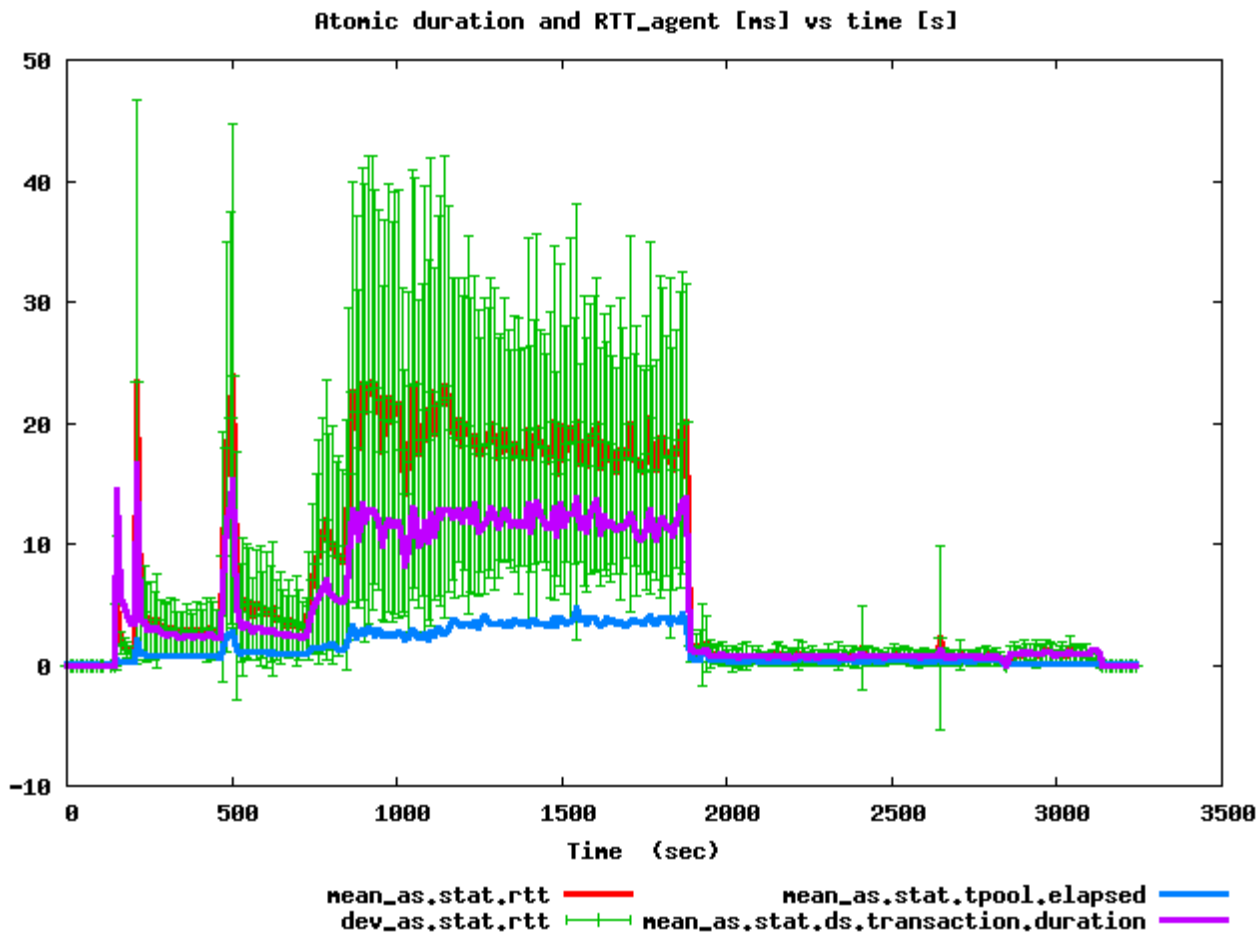


Figure 4.8 – Valeur moyenne des sondes et du RTT calculé grâce à une moyenne exponentielle adaptée. Les valeurs statistiques sont réalisées sur des intervalles de dix secondes et la déviation du RTT est représentée par les barres verticales. Les messages sont les SUBSCRIBE GROUP à taux variable présenté dans la section 6.3.2.2.

Les temps de réponse perçus par les clients qui émettent les requêtes sont des informations importantes concernant la charge du système, car plus leurs valeurs sont importantes et plus le système est chargé. Le tableau 4.10 donne la répartition des temps de traitement des messages et montre que l'utilisation de la moyenne exponentielle adaptée les réduit.

4.3.4.2 La fonction développée à Alcatel-Lucent

L'objectif de cette fonction est de combler les défauts des autres approches pour que le RTT représente la charge d'un agent en prenant en compte l'état de ses voisins. La durée de toutes les interactions asynchrones est analysée selon une fonction qui répond à trois critères :

1. elle prend en compte les interactions limitantes du système, c'est la prise en compte du goulot d'étranglement de l'agent.
2. elle prend en compte les fréquences d'exécution des interactions asynchrones pour ne pas ignorer ou sous-estimer l'effet des interactions les moins fréquentes. Par exemple si 1000 threads sont exécutés avec des temps de 5 ms et dans ce même laps de temps 100 transactions distribuées se terminent avec des temps de 10000 ms, alors la fonction doit freiner le trafic suffisamment fort.
3. elle supprime les données parasites pour ne pas ralentir le trafic. Il arrive que certaines valeurs soient anormalement élevées en raison de petits problèmes très ponctuels comme la perte d'une connexion ou de bugs.

La fonction décrite par l'algorithme 2 consiste à calculer en permanence la moyenne mobile *currentAvg* (avec *computeAvg()*) des valeurs maximales des interactions asynchrones. La moyenne exponentielle permet de ne pas donner trop d'importance aux valeurs parasites et le fait de prendre les valeurs maximales permet à tout moment de prendre en compte l'interaction limitante. Un mécanisme de sécurité est implémenté pour que le RTT diminue dans le cas où les durées des interactions asynchrones diminuent, la moyenne est décrémentée de DECREASE_PERCENT toutes les 100 ms. [3]

La constante TOLERANCE_PERCENT a été ajouté plus tard, elle permet d'être moins stricte sur la prise en compte du maximum des valeurs. Plus elle est élevée et plus cette fonction se comporte comme une moyenne exponentielle. Sa valeur a été fixée à 50% de manière arbitraire. L'ajout de la constante TOLERANCE_PERCENT est difficilement justifiable et ne semble pas très utile, car la fonction fonctionnait très bien sans.

Chaque fois que le token entre dans l'agent, il y est retenu pendant *currentAvg* millisecondes. Ce temps de rétention est mis à jour par la fonction *setResponseTime(currentAvg)* chaque fois que la moyenne est modifiée par la prise en compte de nouvelles valeurs.

Algorithme 2 La fonction de calcul du RTT.

```
1: newValue ← threadElapsed|transactionElapsed|otherOperation
2: if newValue > (currentAvg − currentAvg ∗ TOLERANCE_PERCENT) then
3:     currentAvg ← computeAvg(currentAvg, newValue) ;
4:     setResponseTime(currentAvg) ;
5:     time ← getTime() ;
6: else if (time − getTime()) > 100 then
7:     currentAvg ← currentAvg − currentAvg ∗ DECREASE_PERCENT ;
8:     setResponseTime(currentAvg) ;
9: end if
```

À ce stade, il est légitime d'hésiter entre la moyenne exponentielle adaptée et cette fonction. Les tests suivants permettent de les départager et de choisir la fonction développée à Alcatel-Lucent. Elle permet d'une part de réduire le temps de traitement des messages, car il y a moins d'attente et donc moins de surcharge et d'autre part elle permet de maîtriser fortement la variabilité de RTT (figure 4.9).

Les temps de réponse sont plus courts avec la moyenne exponentielle adaptée par rapport à la moyenne exponentielle car elle donne plus d'importance à l'interaction asynchrone limitante, mais la fonction d'Alcatel-Lucent réduit encore ces temps par rapport à la moyenne exponentielle adaptée (tableau 4.10).

3. Ce délai de 100 ms est arbitraire. La valeur de DECREASE_PERCENT est de 10% et est également arbitraire.

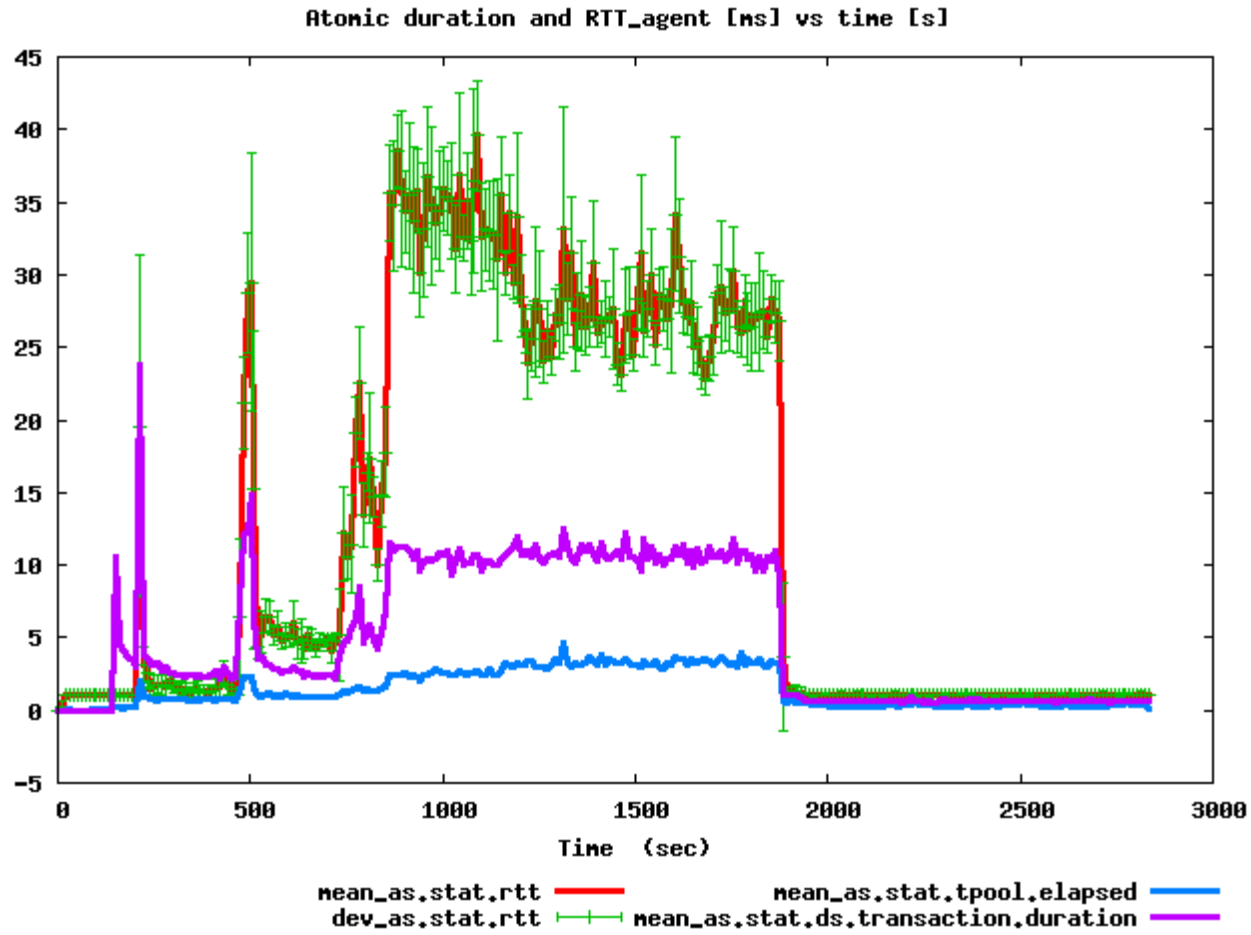


Figure 4.9 – Valeur moyenne des sondes et du RTT calculé grâce à la fonction d'Alcatel-Lucent. Les valeurs statistiques sont réalisées sur des intervalles de dix secondes et la déviation du RTT est représentée par les barres verticales. Les messages sont les SUBSCRIBE GROUP à taux variable présenté dans la section 6.3.2.2.

Temps de traitement [ms]	moyenne exp [msg]	moyenne exp adaptée [msg]	fonction Alcatel-Lucent
$0 < 100$	38580	41915	46842
$100 < 200$	21510	18369	14708
$200 < 300$	2738	2027	1072
$300 < 400$	140	79	57

Figure 4.10 – La répartition des temps de traitement des messages en tranche de 100 ms pour les trois fonctions testées. Les messages sont les SUBSCRIBE GROUP à taux variable présenté dans la section 6.3.2.2.

4.4 Conclusion

Le contrôle d'admission est géré par deux algorithmes, le premier est l'algorithme de contrôle de flux et le deuxième est l'algorithme d'évaluation de la charge globale du système. Le premier prend la décision de transmettre ou de refuser des messages tandis que le second lui envoie des informations sur la charge globale de la plateforme. Ces algorithmes sont intimement liés à l'algorithme du partage de la charge.

Le suivi de l'ASR est particulièrement bien développé, spécialement grâce au metering service. Les données brutes qu'il fournit sont traitées par une fonction d'analyse statistique, elle possède la responsabilité de représenter au mieux la charge de l'agent et de ses voisins. Le temps calculé par cette fonction est le temps de rétention du token dans l'agent, il représente une contribution très importante au RTT du token. Trois fonctions sont testées et celle d'Alcatel-Lucent semble être la plus apte à évaluer la charge de la plateforme.

Chapitre 5

La modélisation

La modélisation est une pratique courante dans les activités scientifiques et technologiques et elle est largement utilisée dans le champ multi-disciplinaire qu'est l'informatique. La compréhension de la notion de modèle et de sa vérité est fondamentale.

Selon Stachowiak, les modèles possèdent trois fonctions principales [14].

La fonction de représentation. Un modèle modélise toujours de quelque chose. Plus un système est complexe, plus il est intéressant de le modéliser, car il est décrit comme un jeu d'éléments dont les interrelations sont clairement mises en évidence. Charniak et McDermott (1985) définissent la représentation comme "une version stylisée du monde" [12] car bien souvent les représentations de système complexe sont plus simples et élégantes.

La fonction subjectivante. Un modèle est toujours construit à partir d'un système pour répondre à des questions précises le concernant. On modélise toujours la réalité pour l'instrumentaliser par la suite. Un modèle doit toujours être interprété en accord avec son but et son usage.

La fonction de réduction. Un système est souvent complexe et il est impossible de le représenter dans son entièreté. Seuls certains critères et paramètres du système sont pris en compte, car ils sont en relation avec la résolution de l'objectif.

La vérité est la propriété d'un énoncé qui porte sur la réalité, elle se définit en fonction de critères de vérités tels que l'adéquation de la connaissance à son objet. À ce sujet, Kant affirme que "la vérité, dit-on, consiste dans l'accord de la connaissance avec l'objet" [17].

Nous devons être conscients de la complexité infinie du monde et de la limite des modèles. Ils ne présentent qu'une partie de la réalité en fonction d'un objectif. Ainsi, on ne connaît jamais la vérité, elle ne peut être qu'à tout moment approchée. Einstein, connu pour sa théorie de la relativité, a introduit la notion de vérité absolue et de vérité relative. Il explique qu'on ne connaîtra jamais la vérité absolue, mais qu'on peut s'en approcher en modélisant de mieux en mieux le réel, chaque version d'un modèle présentant une vérité relative. Il explique qu'une première équation peut donner des résultats confirmés expérimentalement dans un cas, mais pas dans un autre. L'équation est alors améliorée pour en donner une deuxième confirmant les résultats dans les deux cas. Les deux équations sont vraies, mais la deuxième l'est plus que la première, car elle s'applique à un domaine de réalité plus large. On peut considérer que la deuxième formule est encore une approximation d'une formule plus générale [16].

Selon George Box, "tous les modèles sont faux [. . .]" . Un modèle est toujours faux dans le sens où il n'est toujours qu'une représentation approximative de la réalité, mais ils sont tous vrais dans leur domaine d'application. On insiste sur le fait qu'il est extrêmement important de cerner les limites d'un modèle à partir de ses hypothèses , de son objectif, et de son contexte avant de l'utiliser.

Karl Popper manipule également le concept de la vérité pour établir si un modèle est scientifique avec son principe de réfutabilité [24]. Un modèle est réfuté, c'est-à-dire qu'il est faux, lorsqu'il échoue dans la prédiction de faits qui se situent dans son champ d'application. Il est nécessaire de cerner les limites du modèle à partir de ses hypothèses simplificatrices

et de contexte, de générer des prédictions et de faire des expériences pour les vérifier. L'expérience est le seul moyen de véritablement vérifier un modèle, car elle a le statut d'un rapport à une extériorité (extériorité signifie qui est en dehors de l'esprit). Le sujet perçoit la réalité via sa théorie et ses modèles, mais ne la contrôle pas. Le sujet fait une expérience en interrogeant la réalité, il ne la maîtrise pas.

Un modèle informatique, construit pour répondre à un besoin, est vrai dans son champ d'application si toutes ses prédictions sont vérifiées expérimentalement. On gardera cependant à l'esprit qu'il est impossible de montrer qu'un modèle est vrai, par contre on peut facilement montrer qu'il est faux. Il est par conséquent nécessaire de réaliser beaucoup de tests pour acquérir la conviction de la véracité d'un modèle.

La construction d'un modèle répond au raisonnement inductif et déductif. Le premier se base sur une observation objective du réel et de l'établissement d'hypothèses, et le deuxième permet de générer des résultats de prédiction qui peuvent être testés. Autrement dit, la conception d'un modèle part du réel et retourne vers le réel. Si une de ces deux phases est mal conduite, alors le modèle produit est faux, c'est-à-dire qu'il n'existe pas ou peu de correspondance entre la réalité et le modèle.

Dans la phase d'induction, l'observation doit être objective et devrait primer sur les autres sources d'information. Les dangers sont les biais d'observation, l'argument d'autorité, la subjectivité, le langage, l'implémentation abusive de valeur, l'utilisation de modèle préconfiguré et de concept non fondé.

Ce chapitre présente un modèle de l'ASR et de l'algorithme de contrôle de flux. En raison de leurs complexités, nous avons simplifié la réalité en posant des hypothèses fortes (section 5.1.1) pour répondre à des questions précises (voir les objectifs dans la section 5.1). Les prédictions de ce modèle sont exprimées dans la section 5.3.4 et une étude de cas est présentée dans le chapitre 6.

5.1 Les objectifs de la modélisation

Un modèle répond toujours à un objectif, le nôtre est de comprendre certains points de l'algorithme à fenêtre (section 3.3). Il est important de comprendre l'utilité et l'impact sur le fonctionnement de la plateforme des différents paramètres.

Concrètement, les modèles doivent

1. expliquer l'effet de la fenêtre sur le temps de traitement des messages.
2. donner une explication sur la motivation des valeurs des paramètres $0 \leq W \leq 60$ et seuil $= 50$ ms. [1]
3. déterminer s'il est utile de modifier la valeur des paramètres voire de s'en passer.

5.1.1 Les hypothèses communes aux modèles

Les principes de la plateforme sont décrits dans le chapitre 2 et le RTT tel qu'il est mesuré dans la plateforme est présenté dans la section 3.4.1.3. [2]

L'ASR est fortement simplifié en accord avec les objectifs de la modélisation. Les hypothèses de modélisation sont présentées ci-dessous (figure 5.1).

5.1.1.1 La simplification de la variabilité

La modélisation de l'arrivée du trafic. L'arrivée de Poisson est le modèle le plus largement appliqué pour modéliser les processus d'arrivée et il est le plus malléable mathématiquement. Le trafic entrant est modélisé par une distribution de Poisson à taux constant. Cette hypothèse de trafic est très simplificatrice, car le trafic Internet, caractérisé par des rafales, est bien moins régulier qu'un processus de Poisson.

1. Pour rappel, W représente la longueur d'une fenêtre. Plus elle est grande, plus le nombre de messages non confirmés dans le système est grand. Le seuil à 50 ms est le paramètre qui permet d'augmenter ou de diminuer la longueur de la fenêtre. Chaque fois qu'un token est receptionné par l'io_handler, il mesure son RTT et le compare à ce seuil. Si le RTT$>$seuil alors la fenêtre est décrémentée, sinon elle est incrémentée.
2. L'expression est sans doute encore plus complexe, c'est aussi, en quelque sorte, un modèle.

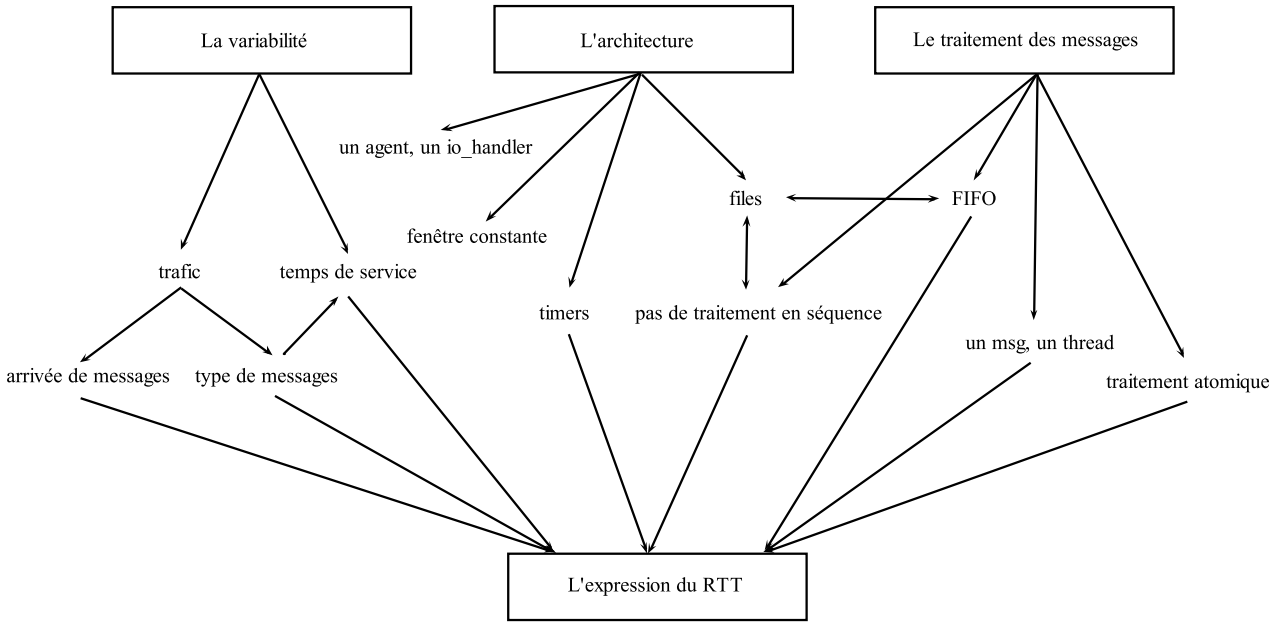


Figure 5.1 – Les hypothèses du modèle. Le sens des flèches donne un sens de lecture et peut être interprété comme un graphe de dépendance entre les critères.

La modélisation du type de messages. On considère dans le modèle un type unique de messages. La réalité est plus complexe, car l'ASR doit faire face à une grande diversité de messages.

Les temps de service. On considère qu'ils sont constants. Le temps de service est assimilé au temps nécessaire à l'ASR pour traiter un message. Normalement ce temps est variable en fonction du type de messages, certains nécessitent un traitement plus long que d'autres. Il est également possible qu'il y ait une variabilité du temps de traitement pour le même type de messages, par exemple en fonction des ressources CPU disponibles à un moment donné.

5.1.1.2 La simplification de l'architecture

La relation un io_handler pour un agent. Le modèle simplifie fortement l'architecture, car d'une part seuls deux composants proxy de la plateforme sont représentés, un agent et un io_handler. [3] D'autre part, la multiplicité de la relation entre un io_handler et un agent est simplifiée à un pour un. La réalité est une relation multiple d'un io_handler pour N agents.

La charge induite dans le temps. Le modèle ne prend pas en compte les charges induites par des tâches programmées dans le temps par le mécanisme des timers. Chaque message envoyé est traité pendant $\frac{1}{\mu}$ ms immédiatement après avoir séjourné dans l'unique file d'attente. En réalité, l'envoi d'un message peut induire une charge résiduelle et périodique longtemps après sa réception dans l'ASR. [4]

Les files d'attente. L'ASR est un système à files d'attente comme expliqué dans la section 2.2 où le modèle Reacteur/-Proacteur indiquait qu'il était nécessaire d'implémenter au moins une file de tâches par réacteur. Le modèle considère uniquement une file pour toute la plateforme (voir la section 2.2).

L'entrée et la sortie des messages à travers le réseau, Linux et le ou les réacteurs, ne sont pas pris en compte (dans le modèle). Ce parcours est considéré comme instantané.

3. Tous les mécanismes de monitoring, gestion de la redondance, etc. ne sont pas représentés alors qu'ils infligent une charge à la plateforme en fonction du trafic.

4. Par exemple, un message M peut armer un timer (voir section 2.2.4) qui toutes les x secondes induit l'envoi d'un message à un composant. Le message M produit une charge résiduelle et périodique après son traitement.

La file est FIFO. On considère le traitement des messages comme étant FIFO, c'est-à-dire que le token ne peut pas dépasser les messages le précédant. En réalité, même si toutes les files de l'ASR sont FIFO, le token peut revenir quatre ou cinq fois plus vite qu'un message nécessitant des accès à une base de données et/ou l'exécution de transactions distribuées.

La longueur W de la fenêtre est fixée. La longueur de la fenêtre de l'io_handler est fixée. En réalité, elle est variable.

5.1.1.3 La simplification du traitement des messages

Le traitement séquentiel n'est pas représenté. La simplification des files d'attente de l'ASR conduit à ignorer les parties de traitement séquentiel d'un message. Seules les parties de traitement parallèle sont considérées, elles représentent les interactions asynchrones exécutées en parallèle dans l'ASR (voir la section 2.2).

Le parallélisme. Le parallélisme est géré par l'utilisation d'un seul thread pool. En réalité l'ASR implémente le pattern Reactor/Proactor (voir la section 2.2.1).

L'atomicité. La décomposition du traitement d'un message en différentes opérations unitaires n'est pas prise en compte. Un message est entièrement traité par un seul thread. C'est une grande simplification, car le réacteur peut programmer plusieurs accès à une base de données, plusieurs transactions distribuées, l'envoi d'autres requêtes, etc. uniquement pour traiter un message. Ces opérations peuvent même être exécutées en parallèle.

5.2 Modélisation *pire des cas* du Dr. Dimitri Tombroff

5.2.1 Les hypothèses supplémentaires

5.2.1.1 La simplification du traitement des messages

Le pessimisme. On considère, lors de l'envoi de W messages, [5] que le traitement de ceux-ci démarre lorsque le dernier message est arrivé dans l'agent. Par conséquent, chaque fois que le token arrive dans l'agent, il y a toujours exactement W messages dans la file d'attente. La réalité est différente, car lorsque le token arrive dans le système, une partie des messages a eu le temps d'être traité. Le RTT est gonflé à sa valeur maximale, car en réalité le token séjourne dans l'agent le temps nécessaire au traitement des X messages restants dans son unique file (X est inférieur à la longueur de la fenêtre). [6]

5.2.2 La présentation du modèle

Cette section est basée sur le document de Dimitri Tombroff placé en annexe. Le modèle consiste à décrire l'ASR comme un io_handler, représenté par une fenêtre de longueur invariable, connecté à un seul agent représenté par une file M/D/S (figure 5.2).

L'agent possède S threads, chacun traite un message en un temps de service de $\frac{1}{\mu}$ secondes. La capacité du système est par conséquent égale à $S\mu$ messages par seconde. Le rôle de la fenêtre est de ne jamais dépasser ce seuil, ou seulement dans certaines situations bien définies. La figure 5.3 illustre le cas où tous les messages sont acceptés, car $\lambda_{\text{inj}} \leq S\mu$. La figure 5.4 illustre le cas contraire où des messages sont refusés. Le voyage du token est représenté par des flèches rectangulaires en pointillés dont la longueur indique la valeur du RTT.

On définit la fenêtre minimale, W_{min} égale au nombre de threads de l'agent ($W_{\text{min}} = S$), car si la longueur de la fenêtre est plus élevée que le nombre de threads, alors l'agent sera toujours surchargé si $\lambda_{\text{transmis}} > S\mu$. Si la longueur de la fenêtre est plus petite que le nombre de threads, alors l'agent sera toujours sous-utilisé. La figure 5.5 représente une situation de

5. W représente la longueur de la fenêtre.
6. On notera cependant que X est égal à W si le taux d'injection λ_{inj} vaut l'infini.

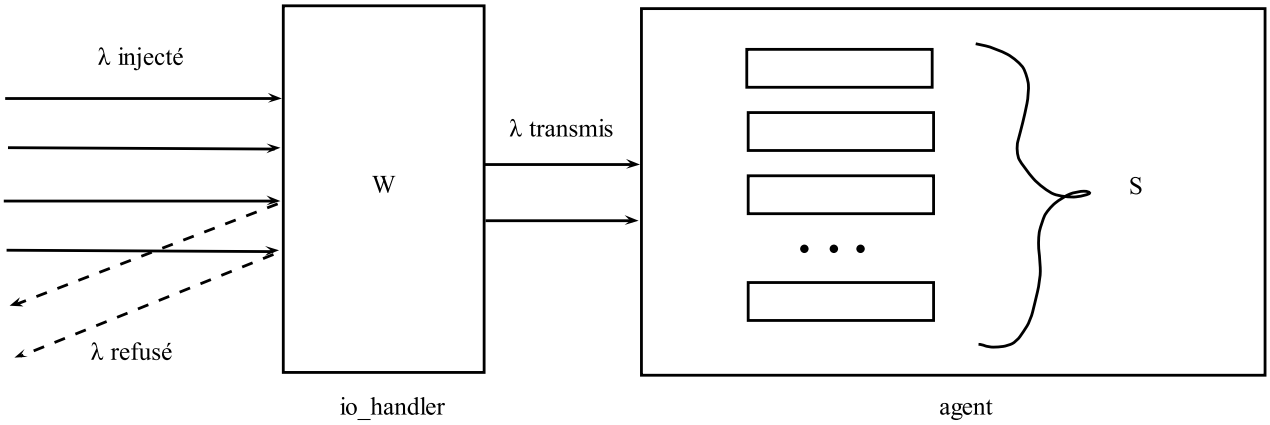


Figure 5.2 – L'io_handler reçoit un trafic à un taux de λ_{inj} messages par seconde, il laisse passer une partie de celui-ci à un taux $\lambda_{transmis}$ et refuse le reste à un taux λ_{ref}.

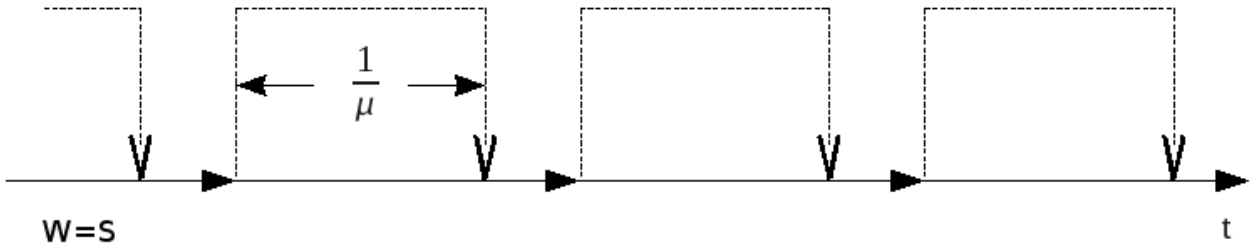


Figure 5.3 – L'agent n'est pas surchargé. Tous les messages sont acceptés.

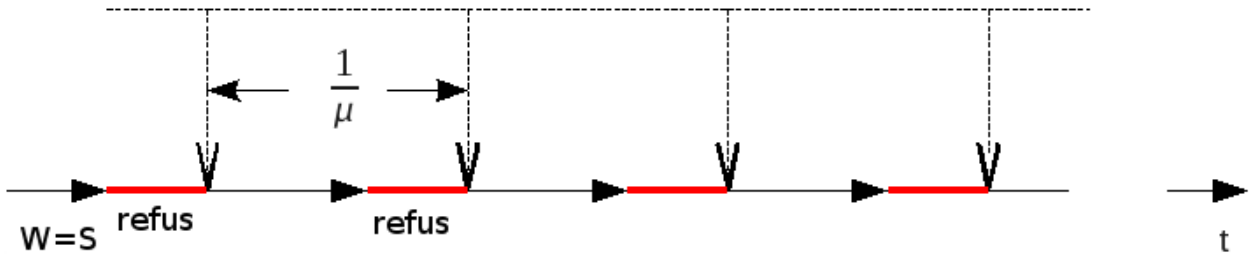


Figure 5.4 – L'agent n'est pas surchargé, car un nombre suffisant de messages est refusé.

surcharge de l'agent, car la fenêtre est plus grande que le nombre de threads disponibles (W>S) et le taux de trafic injecté plus important que la capacité du système ($\lambda_{transmis} > S\mu$) simplement parce que le token prend plus de temps à parcourir sa boucle (car la longueur de la file s'allonge dû au manque de threads pour traiter les messages).

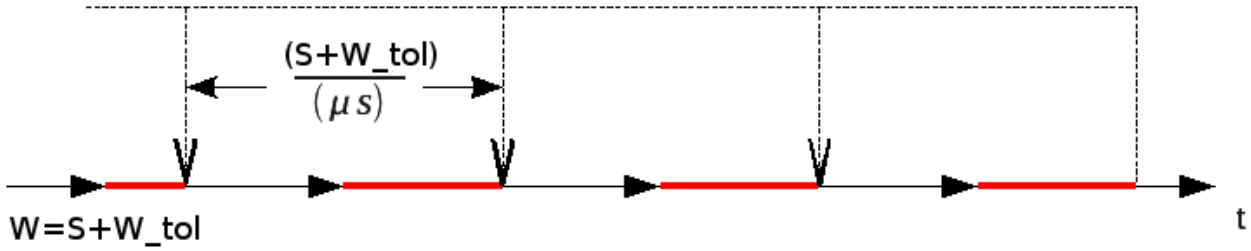


Figure 5.5 – L'agent est surchargé, car un nombre trop peu important de messages sont refusés en raison d'un accroissement de la fenêtre d'une quantité W_tolérée.

Il est possible de déterminer la probabilité de refus d'un message si on considère que l'injection de messages vers l'io_handler suit un processus de Poisson à taux constant de λ_{inj}. L'équation 5.1, l'équation de Poisson, donne la probabilité de recevoir k messages dans un intervalle de t secondes.

$$P(\lambda_{inj}.t, k) = \frac{(\lambda_{inj}.t)^k}{k!} e^{\lambda_{inj}.t} \tag{5.1}$$

Le calcul du taux de messages refusés est complexe, car il dépend du RTT. Il varie à la fois en fonction de la longueur de la fenêtre et du taux de messages injecté (équation 5.2 avec $W > S$).

$$\frac{1}{\mu} \leq RTT \leq \frac{W}{\mu.S} \tag{5.2}$$

La variation du RTT indique que l'algorithme de contrôle de flux possède un mécanisme de rétroaction. Si, à un cycle de l'algorithme, l'io_handler laisse passer plus de messages que l'agent ne peut en traiter (W>S), alors le RTT est plus long au prochain cycle, induisant une augmentation de la probabilité de refus d'un message et donc une hausse du taux de refus. L'effet rétroactif provoque une alternance de vagues de refus et d'acceptation de messages qui compliquent la recherche du taux de messages refusés après une longue période de temps (par exemple une journée). Il est impératif de résoudre une chaîne de Markov.

L'effet rétroactif peut être caractérisé par une suite d'états dont les transitions sont tirées selon une certaine probabilité. On définit l'état S_0 si S messages sont transmis à l'agent, S_1 pour S+1 messages, S_n pour S+n messages, etc. Les probabilités de transition dépendent du trafic, de la fenêtre, et de l'état de l'agent. Les états sont représentés par une chaîne de Markov à la figure 5.6. [7]

Le taux de refus moyen est calculé à partir de la distribution de probabilité des états de l'agent en fonction du trafic (équation 5.3).

$$\lambda_{ref} = \lambda_{inj}(\sum_{k=1}^{\infty} P[S_k]) \tag{5.3}$$

Les probabilités de transition définissent une matrice de transition de Markov qui doit être résolue pour connaître la distribution des probabilités des états.

La probabilité de transition vers un état inférieur, par exemple de l'état S_i vers S_0 est égale à la probabilité de recevoir moins de S messages pendant le RTT associé à S_i et i>0, RTT vaut $\frac{S+i}{\mu.S}$ secondes (La probabilité est calculée à l'équation 5.4).

La probabilité de transition vers un état supérieur, par exemple de l'état S_0 vers S_n est égale à la probabilité de recevoir plus de S+n messages pendant le RTT associé à S_0, RTT vaut $\frac{1}{\mu}$ secondes (La probabilité est calculée à l'équation 5.5).

7. Toutes les transitions ne sont pas représentées.

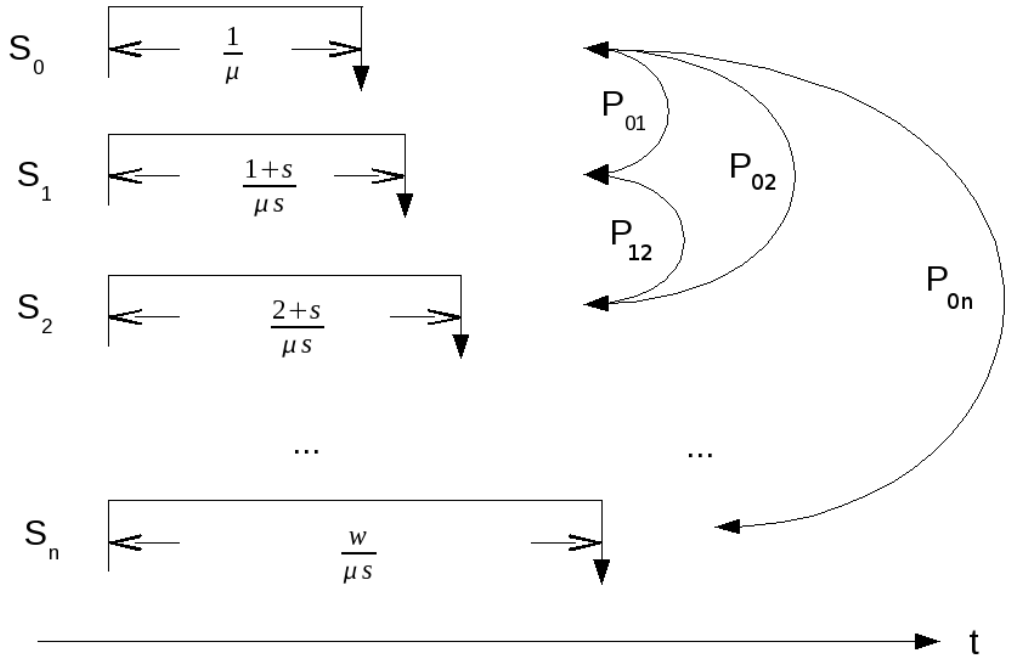


Figure 5.6 – La représentation de l'effet rétroactif par une chaîne de Markov.

$$P_{i0} = \sum_{k=0}^{S} P(\lambda_{inj}.\frac{s+i}{\mu.s}, k) \tag{5.4}$$

$$P_{0n} = \sum_{k=n}^{\infty} P(\lambda_{inj}.\frac{1}{\mu}, k) \tag{5.5}$$

On obtient les équations 5.6 et 5.7 en généralisant les probabilités de transition de i à j et de j à i avec $S \leq i \leq j \leq W$.

$$P_{ij} = \sum_{k=j}^{\infty} P(\lambda_{inj}.\frac{s+i}{\mu.s}, k) \tag{5.6}$$

$$P_{ji} = \sum_{k=i}^{S} P(\lambda_{inj}.\frac{s+j}{\mu.s}, k) \tag{5.7}$$

Le taux de messages transmis est finalement calculé à partir du taux de messages refusés dans l'équation 5.8.

$$\lambda_{transmis} = \lambda_{inj} - \lambda_{ref} \tag{5.8}$$

Ces équations permettent de générer des courbes de résultats présentées et analysées dans la section 5.2.3.

5.2.3 Les prédictions

Les courbes ont été générées par un programme de Dimitri Tombroff (voir annexes). On considère que S=10 et μ vaut 0.1 seconde et le système possède alors une capacité de traitement d'un message par seconde.

La figure 5.7 illustre le RTT moyen du token et la figure 5.8 illustre le taux de messages transmis et refusés. Le nombre de messages transmis varie linéairement pour un taux allant de 0 à 700, la croissance est moins forte au fur et à mesure que le taux s'approche de la capacité de l'agent.

Plus la fenêtre est grande et plus le taux de transmission de messages est important. Le système est largement sous-exploité à W=W_min=S lorsque le taux de messages injectés est équivalent à la capacité de la plateforme. C'est une conséquence de l'écrémage des pics de trafic.

Plus la fenêtre est grande et plus le RTT est grand, car s'ajoutent aux délais de traitement des messages, les délais d'attente dans les files.

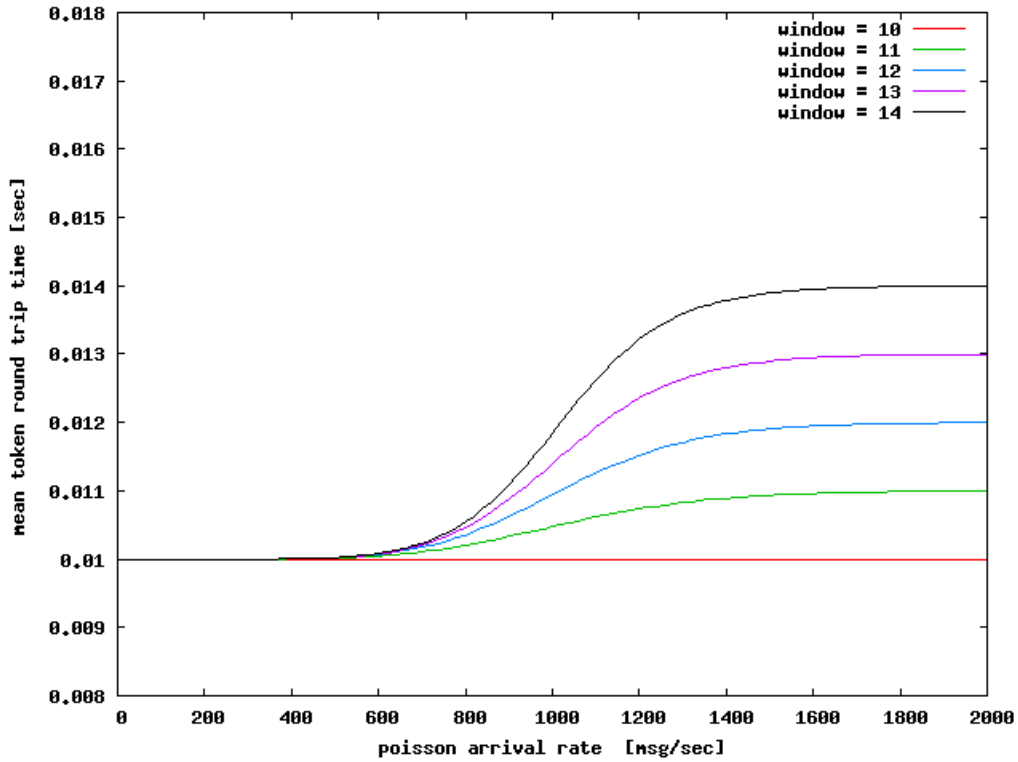


Figure 5.7 – RTT moyen en fonction du taux de messages injectés pour des fenêtres de longueur différente

5.2.4 Conclusion

Les résultats montrent l'effet de la longueur de la fenêtre sur un système basé sur des files d'attente. Premièrement, plus la fenêtre est grande, plus les temps de latence augmentent et moins de messages sont refusés. Deuxièmement, la charge varie linéairement par rapport à la longueur de la fenêtre.

L'étude du temps de voyage du token a été primordiale dans l'établissement de l'équation du taux de refus moyen en fonction du trafic et de la longueur de la fenêtre. Le modèle montre d'une part que la relation entre la longueur de la fenêtre et le RTT moyen est linéaire et d'autre part que le RTT moyen s'exprime selon la variabilité du trafic.

Les objectifs sont remplis, car l'influence du seuil des 50 ms est comprise. L'algorithme de l'ASR diminue la longueur de la fenêtre si l'agent subit une charge trop importante et l'augmente dans le cas contraire. Le modèle à fenêtre fixe montre que la maîtrise de cette variation de W est importante, car elle détermine la qualité du service et le nombre de messages acceptés par unité de temps. L'utilisation de ce seuil ne permet pas de maîtriser ces variations. Par conséquent, on peut le supprimer et le remplacer en ajoutant deux règles simples à l'algorithme.

La première règle indique que si le trafic est plus important que la capacité de l'ASR ($\lambda_{inf} > S\mu$), alors la longueur de fenêtre doit être minimale (W_min).

Deuxièmement, si le trafic est équivalent à la capacité de l'ASR ($\lambda_{inf} = S\mu$), alors la longueur de la fenêtre doit être fixée entre les valeurs W_min et W_min+W_tolérée. L'accroissement de la fenêtre d'une quantité W_tolérée doit être bien maîtrisé, car en cas de saut de trafic supérieur à la capacité du système, une partie sera acceptée en fonction de cette

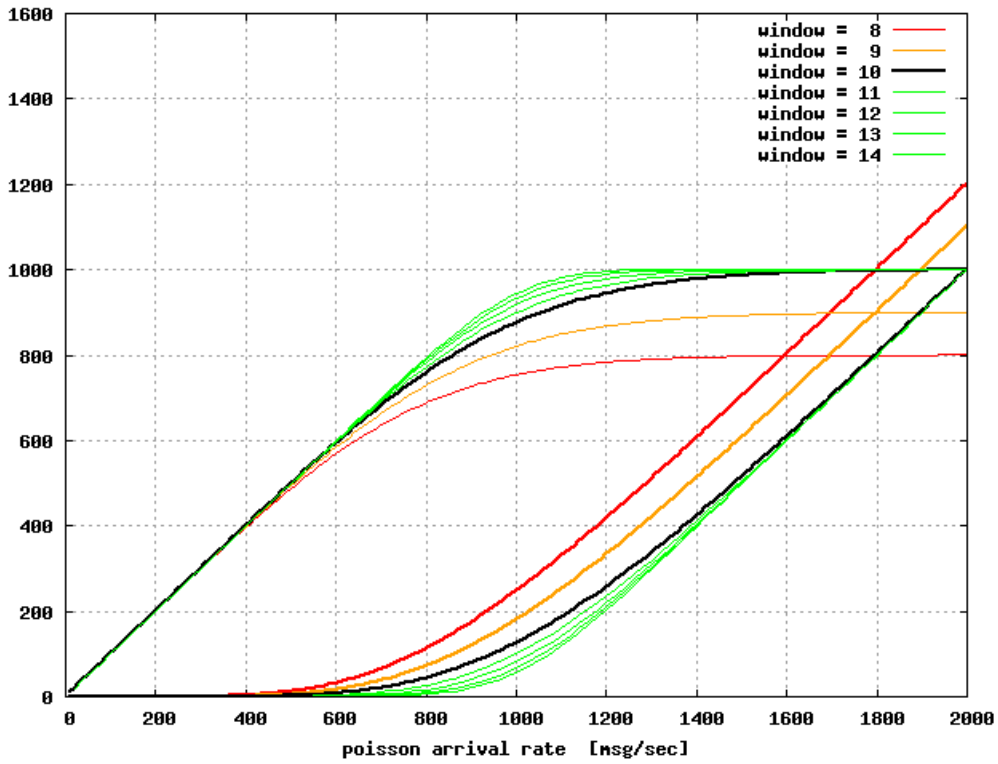


Figure 5.8 – Nombre de messages refusés et transmis par seconde en fonction du taux de messages injectés.

valeur.[8] En cas d'acceptation de surplus de trafic, le délai de traitement des messages est dégradé, celui-ci doit rester acceptable.

5.3 Modélisation *moyenne* de Michaël Bouhy

5.3.1 Les hypothèses supplémentaires

Le traitement des messages Les messages sont traités par l'agent sans désemparer, aucun temps mort n'apparaît entre deux d'entre eux. L'agent ne prend pas de pause dans le traitement des messages.

5.3.2 La présentation du modèle

Le modèle consiste à décrire l'ASR comme un io_handler, représenté par une fenêtre de longueur invariable, connecté à un seul agent représenté par une file M/D/S (figure 5.2).

La variation du RTT indique que l'algorithme de contrôle de flux possède un mécanisme de rétroaction. Si l'io_handler laisse passer plus de messages que l'agent ne peut en traiter (W>S), alors le RTT est plus long. L'effet rétroactif est à l'origine d'une alternance de période T_{inj} de vagues de refus et d'acceptation de messages (figure 5.9).

5.3.3 Les équations

Un cycle, défini en fonction de l'état interne de la fenêtre, est le fait de remplir entièrement une fenêtre vide et de revenir vers l'état initial. Les cycles de l'algorithme sont composés d'une période de transmission et d'une période de

8. Si W_tolérée=0, alors tout le pic de trafic est refusé.

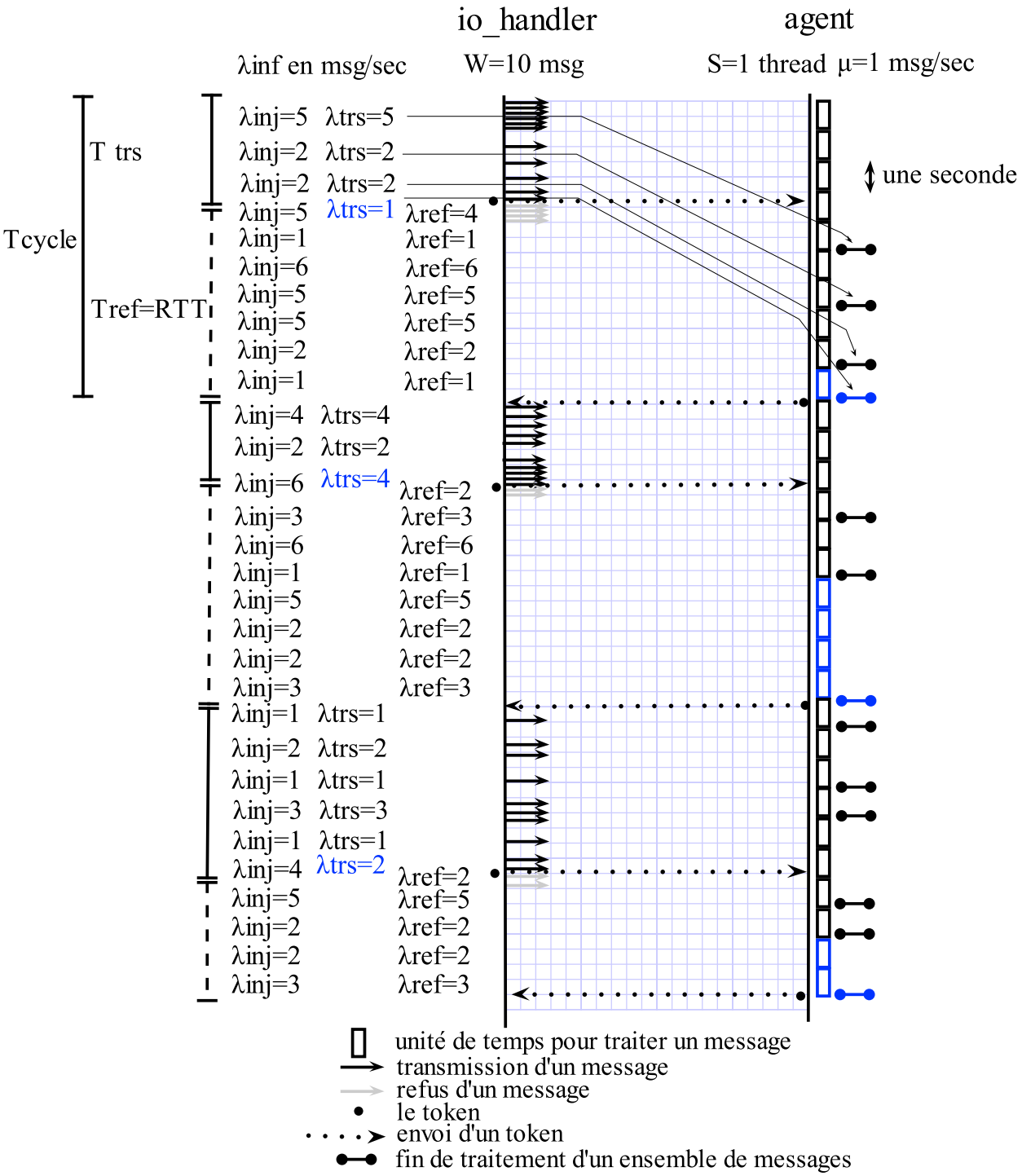


Figure 5.9 – La représentation de l'effet rétroactif avec un diagramme de séquence.

refus de messages par l'io_handler. Les caractéristiques périodiques et indépendantes des cycles permettent d'induire des équations à partir de la figure 5.9.

L'injection de messages est représentée par le processus de Poisson $\overline{\lambda}_{inj} = \forall t : \{\lambda_{inj}(t)\}$. La distribution des taux de transmission ou de refus pendant les périodes correspondantes[9] n'a aucune influence sur le RTT et par conséquent sur le nombre de messages refusés. C'est pour cette raison qu'il est permis de travailler avec des moyennes statistiques des taux de messages.

Au vu des premiers cycles de la figure 5.9, il est clair que l'algorithme refuse des suites consécutives de messages. Pendant le premier cycle, 10 messages sont transmis pendant T_{trs} secondes à un taux moyen de $E(\lambda_{inj(t)})$ et 24 messages sont refusés pendant T_{ref} secondes au taux moyen de $E(\lambda_{inj(t)})$.

On définit la période d'un cycle de l'algorithme par la grandeur de temps T_{cycle}, elle représente le temps d'injection de messages. Une partie de ceux-ci sont transmis vers l'agent pendant T_{trs} unités de temps et l'autre partie est refusée pendant T_{ref} unités de temps.

$$T_{cycle} = T_{trs} + T_{ref} \tag{5.9}$$

Les N_{ref} messages refusés et les N_{trs} messages transmis sont comptés respectivement pendant les périodes de refus T_{ref} et de transmission T_{trs}. Dans ce modèle, les messages ne se perdent pas et ne sont pas créés dans le système (équation 5.10).

$$N_{cycle} = N_{ref} + N_{trs} \tag{5.10}$$

Le temps de transmission correspond au temps nécessaire pour remplir une fenêtre entièrement. Il est calculé par l'équation 5.11. Cette grandeur ne dépend que de la moyenne du taux d'injection et de la longueur de la fenêtre.

$$T_{trs} = \frac{W[msg]}{E(\lambda_{inj}(t))[\frac{msg}{sec}]}[sec] \tag{5.11}$$

Le temps de refus correspond au RTT du token, celui-ci est envoyé lorsque la fenêtre est pleine de messages. L'équation 5.12 montre que le RTT correspond au temps nécessaire pour traiter le message le précédant au moment où le token rentre dans l'agent.[10]

$$T_{ref} = RTT = W(\frac{1}{S\mu} - \frac{1}{E(\lambda_{inj}(t))}) \tag{5.12}$$

A partir des équations 5.9 et 5.11, on montre grâce à l'équation 5.13 que la période du cycle est toujours constante, elle ne dépend pas du trafic injecté ! La période du cycle est d'autant plus grande que l'agent est lent et que la fenêtre est grande.

$$T_{cycle} = T_{trs} + T_{ref} = \frac{W[msg]}{S\mu[\frac{msg}{sec}]}[sec] \tag{5.13}$$

L'objectif est de déterminer le nombre de messages transmis N_{trs} et refusés N_{ref} après une longue période de temps pour voir si la fenêtre joue un rôle important dans l'expression des taux moyens de transmission et de refus de messages. L'exclusivité des périodes de refus et de transmission permet de calculer N_{ref} et N_{trs} pendant un cycle de l'algorithme (équations 5.14 et 5.15).

$$N_{ref} = E(\lambda_{inj}(t))T_{ref}[msg] = W(\frac{E(\lambda_{inj}(t))}{S\mu} - 1) \tag{5.14}$$

$$N_{trs} = E(\lambda_{inj}(t))T_{trs}[msg] = W \tag{5.15}$$

9. la période de transmission et de refus
10. Des messages ont eu le temps d'être traités.

Ces deux résultats permettent de calculer le nombre de messages intervenants pendant un cycle de l'algorithme (équation 5.16). Plus le taux d'injection est élevé, plus la fenêtre est grande et plus l'agent est lent, alors plus le nombre de messages intervenants est élevé.

$$N_{cycle} = E(\lambda_{inj}(t))\frac{W}{S\mu} \tag{5.16}$$

On calcule la proportion R_{ref} de messages refusés pendant un cycle (équation 5.17) permettant par la suite la détermination du taux moyen de messages refusés pendant celui-ci (équation 5.19).

$$R_{ref} = \frac{N_{ref}}{N_{ref} + N_{trs}} = 1 - \frac{S\mu}{E(\lambda_{inj}(t))} \tag{5.17}$$

$$\lambda_{ref_{cycle}} = R_{ref}E(\lambda_{inj}(t)) = (1 - \frac{S\mu}{E(\lambda_{inj}(t))})E(\lambda_{inj}(t)) = E(\lambda_{inj}(t)) - S\mu \tag{5.18}$$

$$\lambda_{ref_{cycle}} = E(\lambda_{inj}(t)) - S\mu \tag{5.19}$$

Le taux de refus moyen après un nombre important de cycles est connu, car il est possible de calculer l'espérance et la variance du processus de Poisson $\overline{\lambda}_{inj}$ dont l'espérance est calculée pour un nombre important d'événements et vaut λ_{inj} (équation 5.20).

$$\lambda_{ref} = \lambda_{inj} - S\mu \tag{5.20}$$

L'écart type d'un processus de Poisson $\overline{\lambda}$, de la même manière que pour son espérance, est connue et vaut $\sqrt{\lambda}$ (équation 5.21).

$$\Delta\lambda_{ref} = \pm\sqrt{\lambda_{inj}} \tag{5.21}$$

L'équation 5.22 est vérifiée à tout moment, car les flux de messages sont conservés. Cette équation et l'équation 5.21 permettent de calculer le taux moyen de messages transmis vers l'agent.

$$\lambda_{inj} = \lambda_{trs} + \lambda_{ref} \tag{5.22}$$

$$\lambda_{trs} = \begin{cases} S\mu & \text{si } \lambda_{inj} > S\mu \\ \lambda_{inj} & \text{si } \lambda_{inj} \leq S\mu \end{cases} \tag{5.23}$$

$$\Delta\lambda_{trs} = \pm\sqrt{\lambda_{inj}} \tag{5.24}$$

L'équation 5.23 montre en **moyenne** que l'agent ne reçoit pas plus de messages qu'il ne peut en traiter et cela est indépendant de la longueur de la fenêtre.

5.3.4 Les prédictions

Les équations 5.20 et 5.23 montrent que l'algorithme à fenêtre envoie en moyenne uniquement ce que l'agent peut traiter et refuse les autres messages indépendamment de la longueur de la fenêtre.

La figure 5.10 illustre le comportement du système. Il n'y a, en moyenne, aucun refus jusqu'à 1000 messages par seconde, mais il arrive d'en observer déjà à partir de 990 ou seulement à partir de 1010 messages par seconde. Ce plateau est dû à la variabilité du trafic (qui suit un processus de Poisson).

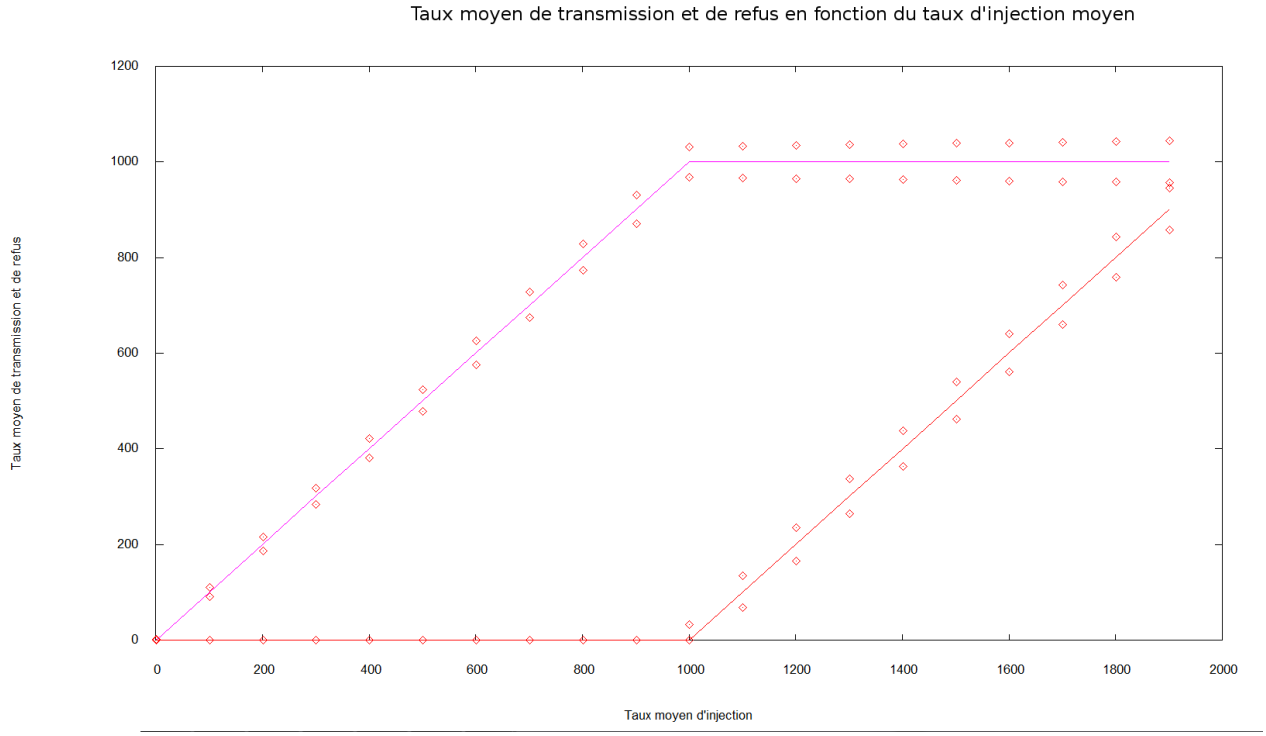


Figure 5.10 – Evolution des taux moyens de transmission et de refus en fonction du taux moyen de messages injectés. Les lignes en pointillé montrent l'écart-type autour des lignes pleines qui représentent des valeurs moyennes.

La longueur de la fenêtre influence d'une part la période des alternances entre refus (équation 5.12) et transmission (équation 5.11) de messages et d'autre part la durée moyenne de traitement d'un message. [11]

La longueur W de la fenêtre influence la tolérance aux pics de trafic. Une fenêtre de longueur 10 avec S=1 et μ=1 indique que le système tolère soit, un pic à $\lambda_{inj} = 1000$ messages secondes pendant $\frac{1}{100}$ ième de seconde, soit $\lambda_{inj} = 100$ pendant $\frac{1}{10}$ ième de seconde, soit $\lambda_{inj} = 10$ pendant une seconde, soit $\lambda_{inj} = 1$ pendant 10 secondes, etc. [12] Les figures 5.11, 5.12 et 5.13 montrent l'influence de la fenêtre sur le traitement des pics de trafic, plus la fenêtre est petite et plus les périodes de refus sont fréquentes et courtes.

Plus la fenêtre est petite, plus le nombre de tokens envoyés est important. Le token a pour fonction de s'assurer que la file de l'agent est vide avant que de nouveaux messages soient envoyés. Plus fréquemment les tokens sont envoyés et plus petite est la file d'attente des messages. La qualité de service est assurée si W<=S.

5.3.5 Conclusion

La fenêtre joue un rôle important, car elle régule le trafic grâce au seul effet rétroactif. Sa longueur n'a pas d'influence sur la quantité de messages transmis ou refusés, mais une longueur plus grande que le nombre de threads de l'agent permet d'accepter plus largement des pics de trafic au dépit de la qualité de service.

La qualité de service est optimale [13] pour une fenêtre de longueur égale ou inférieure au nombre de threads de l'agent, un agent n'est donc jamais sous-exploité ! [14]

Le meilleur choix pour la longueur de la fenêtre est donc W=S, car si elle est inférieure à S, alors un nombre trop important de tokens (inutiles) sont échangés. Une fenêtre plus grande que S surcharge moins le système, car le token est moins utilisé et permet de traiter des variations de trafic plus facilement, mais avec détérioration du temps de service.

11. Le RTT représente la durée maximale de traitement d'un message. Le RTT augmente avec la fenêtre.
12. Avec l'hypothèse que la fenêtre soit vide quand le pic de trafic est injecté.
13. Une qualité optimale signifie qu'aucun message n'attend avant de subir son traitement. Les temps d'attente dans l'agent sont nuls.
14. On pourrait penser que si la longueur de la fenêtre deux fois plus petite que le nombre de threads disponibles, alors la moitié des messages injectés sont refusés. Non, la fenêtre connaîtra un deuxième cycle pour traiter la deuxième partie des messages.

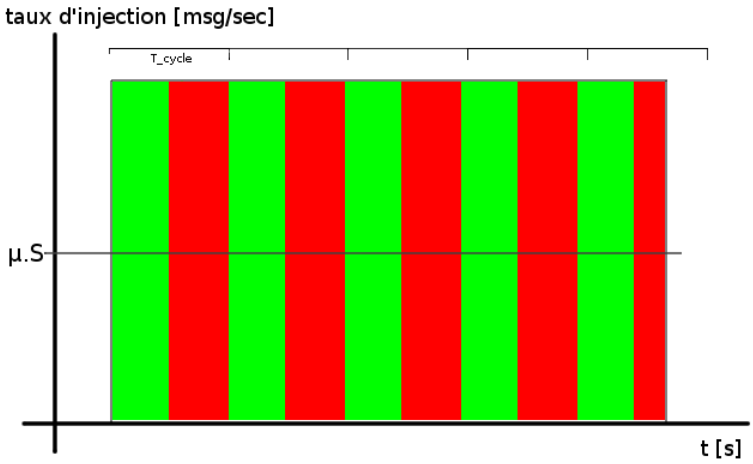


Figure 5.11 – La longueur de la fenêtre est importante, car elle définit la longueur des cycles. Les cycles sont deux fois plus courts que sur la figure 5.12 et quatre fois plus courts que sur la figure 5.13

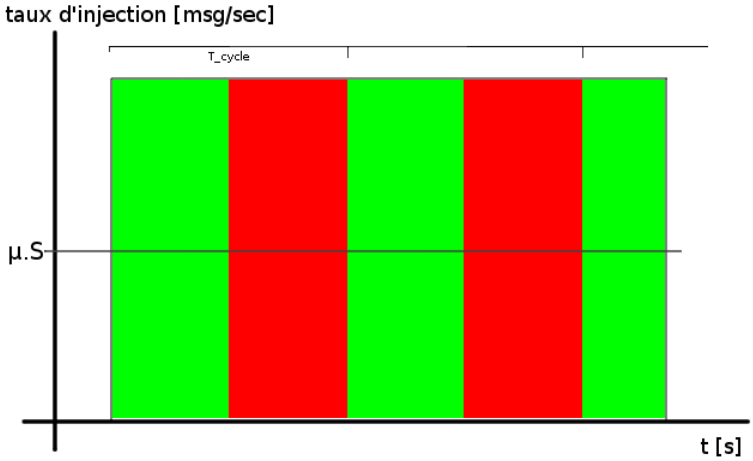


Figure 5.12 – La longueur de la fenêtre est deux fois plus grande que sur la figure 5.11 et deux fois plus petite que sur la figure 5.13

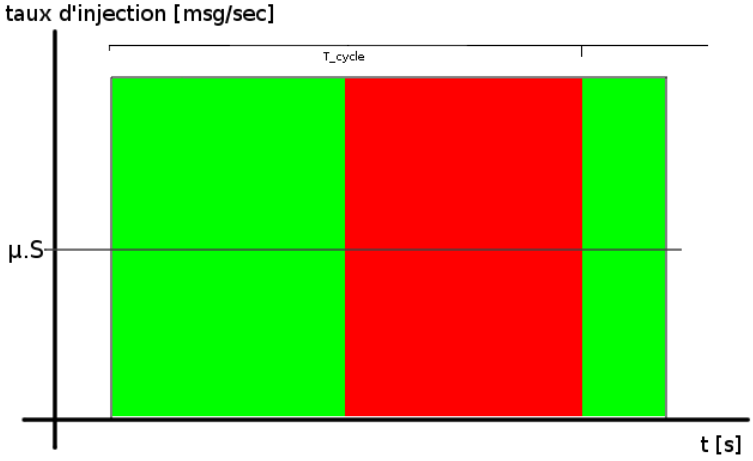


Figure 5.13 – La longueur de la fenêtre est deux fois plus grande que sur la figure 5.12 et quatre fois plus grande que sur la figure 5.11

5.4 Conclusion

Les deux modèles sont très intéressants, car ils partent des mêmes hypothèses simplificatrices et arrivent à des résultats semblables. Il y a cependant un point de discorde très important, le rôle de la longueur de la fenêtre n'est pas compris de la même façon. Le modèle de Dimitri Tombroff prédit après une longue période de test que le taux de refus moyen des messages sera d'autant plus important que la fenêtre est petite, tandis que le modèle de Michaël Bouhy prédit que la longueur de la fenêtre n'influence pas le taux de refus moyen.

Le modèle de Dimitri Tombroff fait appel à des techniques mathématiques plus lourdes que le modèle de Michaël Bouhy, mais le seul moyen d'établir la vérité est de réaliser des tests plus complets et de corriger itérativement les modèles. Dans les deux cas, les modèles mettent en évidence le rôle certain de la fenêtre dans l'admission des messages et montrent que la maîtrise de sa longueur est importante, car elle influence la qualité de service et la gestion des pics de trafic.

Chapitre 6

Une étude de cas

L'étude de cas consiste à montrer l'impact des corrections apportées à l'algorithme de contrôle de flux. La correction permet de mieux évaluer la charge globale du système. L'application Presence a été sélectionnée pour sa complexité car elle utilise un large spectre de fonctionnalités de l'ASR, et elle n'est pas orienté calcul, [1] mais communication.

Le principe du contrôle d'admission et des sondes sont rapidement rappelés. Le service Presence et le déploiement sont présentés en détail avant la présentation des résultats où l'on montre l'impact de l'utilisation des sondes.

6.1 Le contrôle de flux et les sondes

Le contrôle d'admission est réalisé grâce à un token. Il est envoyé tous les W messages vers l'agent et il y est retenu pendant un temps RTT_agent proportionnel à sa charge. L'io_handler refuse tous les messages injectés tant qu'il ne possède pas le token, il représente une permission pour l'io_handler de transmettre W messages vers un agent (figure 6.1).

Le RTT_agent est mesuré dans chaque agent grâce au metering service. Ce dernier permet de placer des sondes dans n'importe quelle section du code pour mesurer son exécution (voir section 2.4). La figure 6.2 montre l'évolution des mesures de chaque sonde ainsi que le RTT_agent calculé grâce à la fonction de traitement développée à Alcatel-Lucent (voir section 4.3.4.2).

L'utilisation des sondes permet d'étendre le champ de vision de l'io_handler frontière, car elle prend en compte les interactions extérieures au groupe Presence Server (figure 6.3). Les interactions sont les accès à la base de données via les threads du thread pool (voir section 2.2.2) et les transactions distribées (voir section 2.2.3).

6.2 Le service Presence

Les messageries en ligne, telles que les chats, possèdent souvent un onglet permettant de modifier son état de présence. Un état peut être soit "en ligne", "occupé", "absent", etc. Les utilisateurs [2] peuvent suivre l'évolution de l'état de présence de leurs amis (si ceux-ci leur permettent de le faire). Toutes ces informations sont gérées par l'application Presence.

L'application Presence collecte les documents de présence créés par un client de messagerie en ligne et notifie les utilisateurs observateurs à propos de leur modification. Différents messages Presence sont échangés, les PUBLISH, les PRESENCE SUBSCRIPTION et les WATCHER INFO SUBSCRIPTION.

Un PUBLISH crée ou modifie un document présence d'un utilisateur. Ce document possède un délai d'expiration et il est automatiquement supprimé si son timer (section 2.2.4) n'a pas été réinitialisé avant son expiration. L'initialisation est réalisée par l'envoi d'un PUBLISH REFRESH.

1. Une application orientée calcul utilise intensivement et en permanence les ressources CPU pour diverses tâches.
2. les observateurs ou autrement dit, les watchers

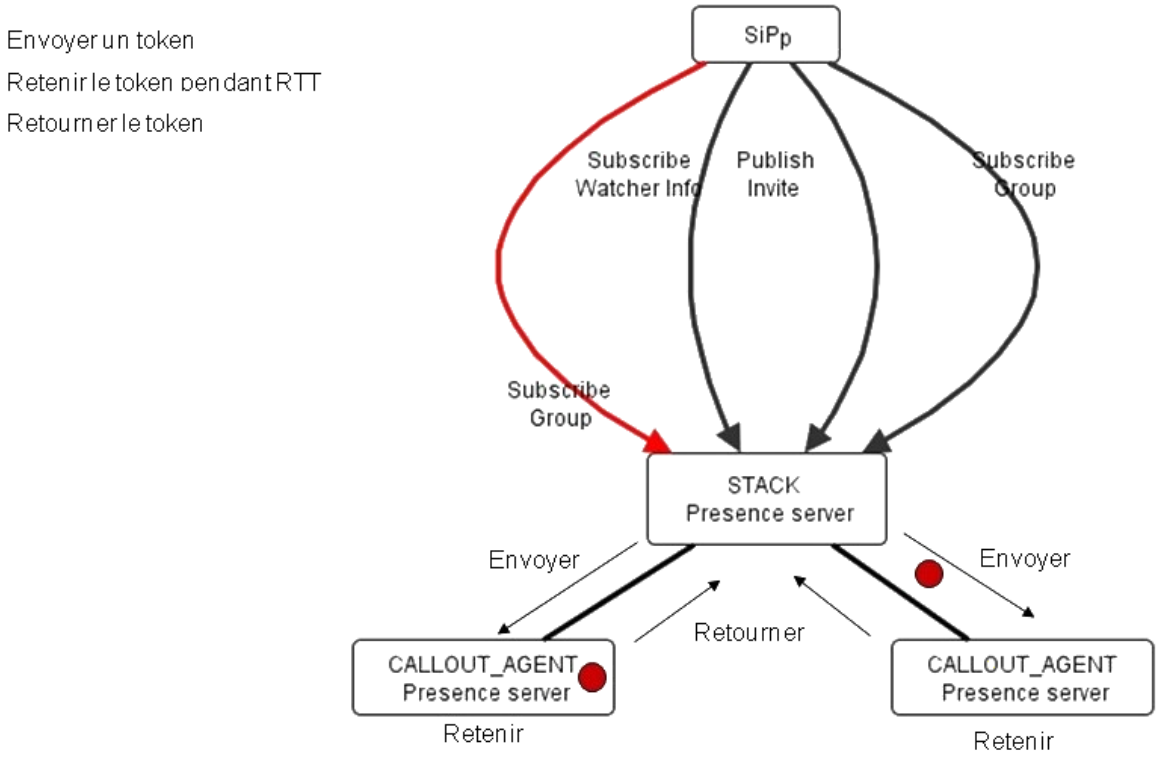


Figure 6.1 – L'algorithme à token consiste à envoyer un token, de le retenir pendant RTT_agent secondes avant de rendre à l'io_handler (ou stack). Ce dernier mesure le temps RTT qui s'écoule entre l'envoi et la réception du token et l'interprète comme une mesure de la charge globale du système. RTT est égale au temps de rétention du token additionné au temps de parcours de celui-ci dans le réseau et les différentes couches protocolaires des composants.

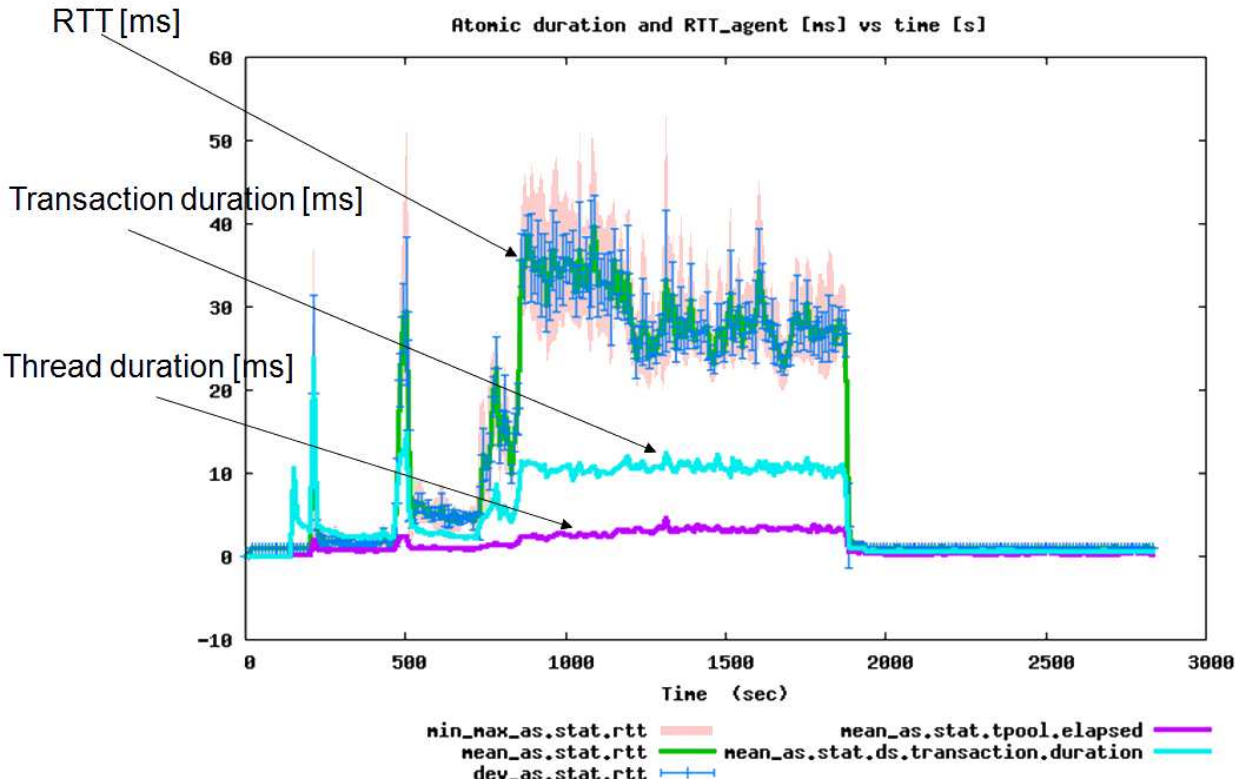


Figure 6.2 – Valeur moyenne des sondes et du RTT_agent. Les valeurs statistiques sont réalisées sur des intervalles de dix secondes.

Un PRESENCE SUBSCRIPTION permet à un utilisateur observateur de demander à être notifié de la présence d'autres utilisateurs. Il reçoit un NOTIFY pour chaque changement d'état d'un de ses amis. Une inscription au système de notification possède un délai d'expiration qui doit être rafraîchi avec un SUBSCRIPTION REFRESH.

Un GROUP SUBSCRIPTION permet de demander à être notifié de l'état de présence d'un ensemble d'utilisateurs.

Un WATCHER INFO SUBSCRIPTION permet à un utilisateur de connaître la liste des utilisateurs qui observent son état de présence.

6.3 L'environnement de test

Cette section présente le déploiement des composants pour le service Presence et la modélisation du trafic.

6.3.1 Le déploiement

Le déploiement des composants importants pour réaliser l'étude de cas est présenté par étape.

SiPp permet d'injecter différents types de trafic SIP dans le système testé selon un scénario précis établi à l'avance, ils sont présentés dans la section 6.3.2. SiPp calcule des données statistiques concernant les appels SIP, il comptabilise le nombre de messages perdus, envoyés, reçus et calcule la moyenne de la durée des appels [26]. Le fait de pouvoir exécuter plusieurs instances de SiPp en parallèle permet d'obtenir des données statistiques pour chaque type de message.

La figure 6.3 montre l'injection du trafic dans le groupe Presence Server. Ce groupe est particulièrement important, car il constitue le point d'entrée du système. Son io_handler (ou stack) est chargé de la difficile mission d'évaluation de la charge globale du système pour prendre l'importante décision de la transmission ou du refus d'un message.

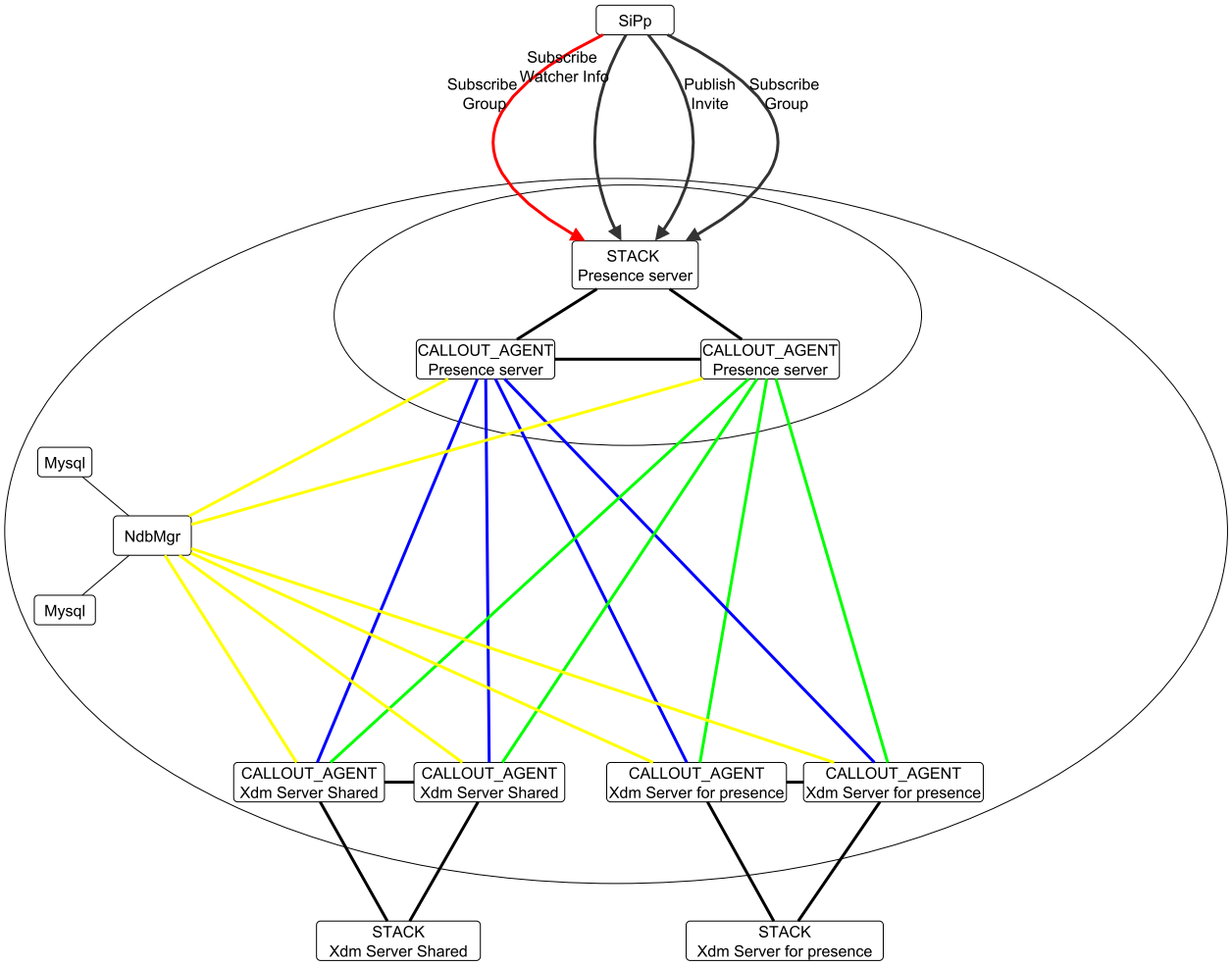


Figure 6.3 – L'injecteur SiPp injecte quatre types de trafic différents dans le groupe Presence Server. Le groupe Presence Server interagit directement avec le groupe de gestion de la base de données. Le groupe XDMS for Presence et shared XDMS gèrent les documents Presence. Le premier est dédié à l'application Presence et le deuxième est partagé par différents serveurs. L'io_handler avec et sans l'utilisation des sondes. Le petit ovale représente le champ de vision de l'io_handler qui n'utilise pas les sondes. Le champ de vision est étendu aux groupes voisins grâce à l'utilisation des sondes et est représenté par le grand ovale.

En réalité, le serveur Presence est déployé sur six agents. Ce nombre important d'agents permet de ne pas réaliser de tests dans un cas particulier de l'algorithme du partage de charge.

Le trafic injecté est divisé en deux parties : la première est un trafic de fond à taux constant et la deuxième est un trafic de stress. Le trafic de fond est constitué de SUBSCRIBE WATCHER INFO, de PUBLISH et de SUBSCRIBE GROUP. Le trafic variable de stress est composé uniquement de SUBSCRIBE GROUP. La variabilité du trafic permet de ne pas tester le système dans un cas particulier.

Les agents du groupe Presence Server sont en communication constante avec la base de données MySQL. Ce groupe est constitué de deux instances de MySQL et d'un manager à qui les requêtes sont destinées. Cette configuration permet le parallélisme du traitement de requêtes (figure 6.3).

Le groupe XDM server for Presence est représenté sur la figure 6.3, il est nécessaire à l'application Presence pour accéder aux données de présence de chaque utilisateur. Ces informations sont stockées dans le réseau sous forme de documents XML, mais ceux-ci sont utilisés et modifiés directement par les utilisateurs. L'application chargée de stocker et de gérer ces documents XML est l'application XDMS (figure 6.3).

6.3.2 La modélisation du trafic

Les quatre types de trafic sont injectés en parallèle selon quatre scénarios présentés de la figure 6.4 à 6.7. Les trois premiers font partie du trafic de fond et le dernier fait partie du trafic de stress.

6.3.2.1 Le trafic de fond

L'injection des messages PUBLISH est décomposée en deux périodes : la première consiste à créer un document Presence pour chaque utilisateur avec des PUBLISH INITIAL et la deuxième permet de réarmer leur timer avant le délai d'expiration de 5 minutes avec des PUBLISH REFRESH. Les deux périodes sont composées uniquement de nouveaux messages, ils peuvent être refusés par le contrôle d'admission si le système est surchargé (réponse 503).

Le refus d'un message PUBLISH a pour conséquence la destruction du document Presence. Son timer n'est pas réarmé et lorsque celui-ci expire, l'application Presence supprime le document. [3]

Le mode Retry-After des messages 503 est activé. Il permet aux clients de créer un nouveau document Presence, pour remplacer celui qui a expiré, grâce à la réémission d'un message PUBLISH.

Le trafic qui réarme les timers des documents est constitué des messages PUBLISH et est en concurrence avec le trafic de stress. Le nombre de refus de PUBLISH (messages 503) devient plus important lorsque le trafic de stress surcharge le système. Un refus, représenté par un message 503 error dans le protocole SIP, entraîne l'expiration d'un document Presence. Les observateurs de ce document sont avertis de sa suppression par l'envoi d'un NOTIFY.

Une minute après avoir démarré l'injection des PUBLISH, l'injection des messages SUBSCRIBE GROUP commence (figure 6.5). Ils permettent aux utilisateurs d'observer les modifications des documents de présence précédemment créés. Ce trafic est divisé en deux périodes, la première permet de créer une inscription pour chaque utilisateur, et la deuxième permet de les rafraîchir avant la fin du délai d'expiration. La première période est composée de nouveaux messages, mais la deuxième ne possède que des messages subséquents, ce sont des messages qui interviennent dans un même dialogue, ils ne doivent pas subir de refus (voir section 3).

Le trafic de fond est complété avec l'envoi de SUBSCRIBE WATCHER INFO pour rendre le test plus général. Il démarre une minute après le début de l'injection des SUBSCRIBE GROUP. Il est également composé de deux périodes, la première possède des nouveaux messages et la deuxième des messages subséquents.

En cas de surcharge, la deuxième période des SUBSCRIBE GROUP et des SUBSCRIBE WATCHER INFO ne peut jamais souffrir d'un seul refus de la part de l'algorithme de contrôle de flux, car elle n'est composée que de messages subséquents, et le nombre de NOTIFY doit augmenter sur le graphique des SUBSCRIBE GROUP.

3. On laisse expirer le document Presence tout seul.

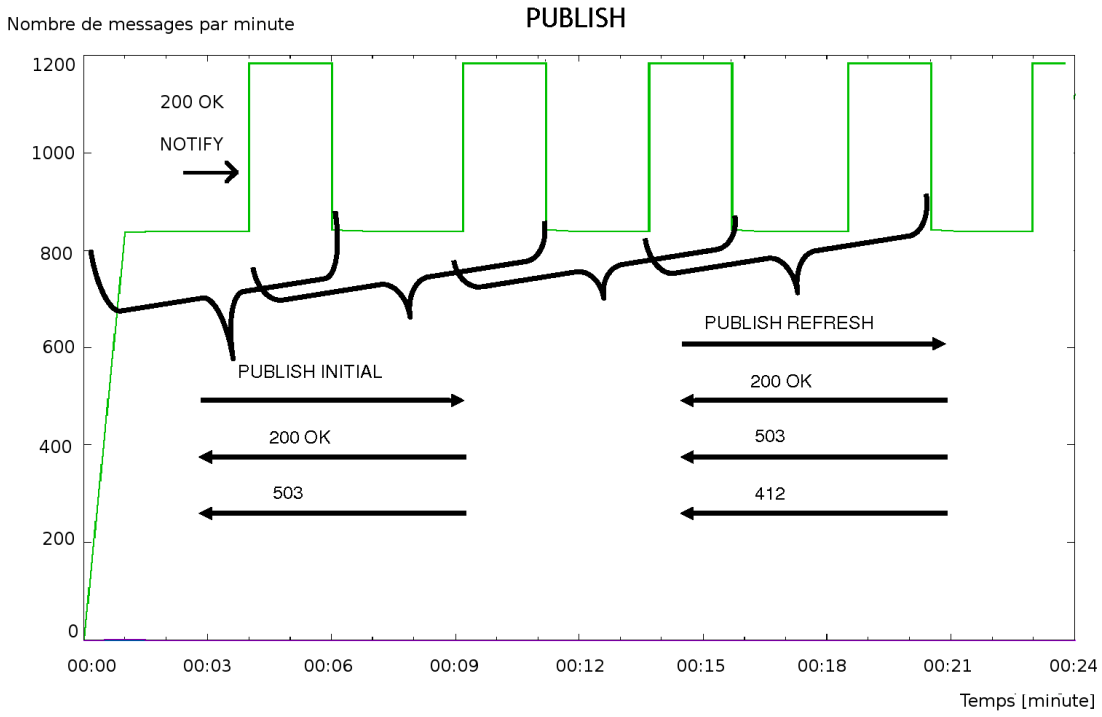


Figure 6.4 – L'injection des messages PUBLISH à taux constant.

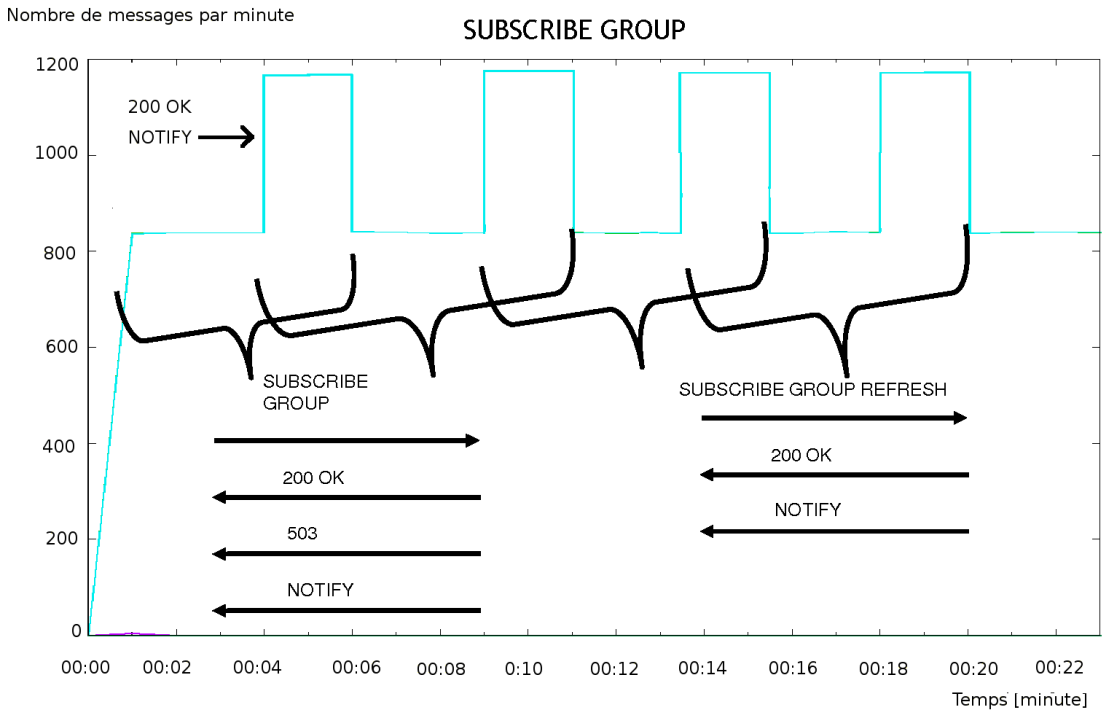


Figure 6.5 – L'injection des messages SUBSCRIBE GROUP à taux constant

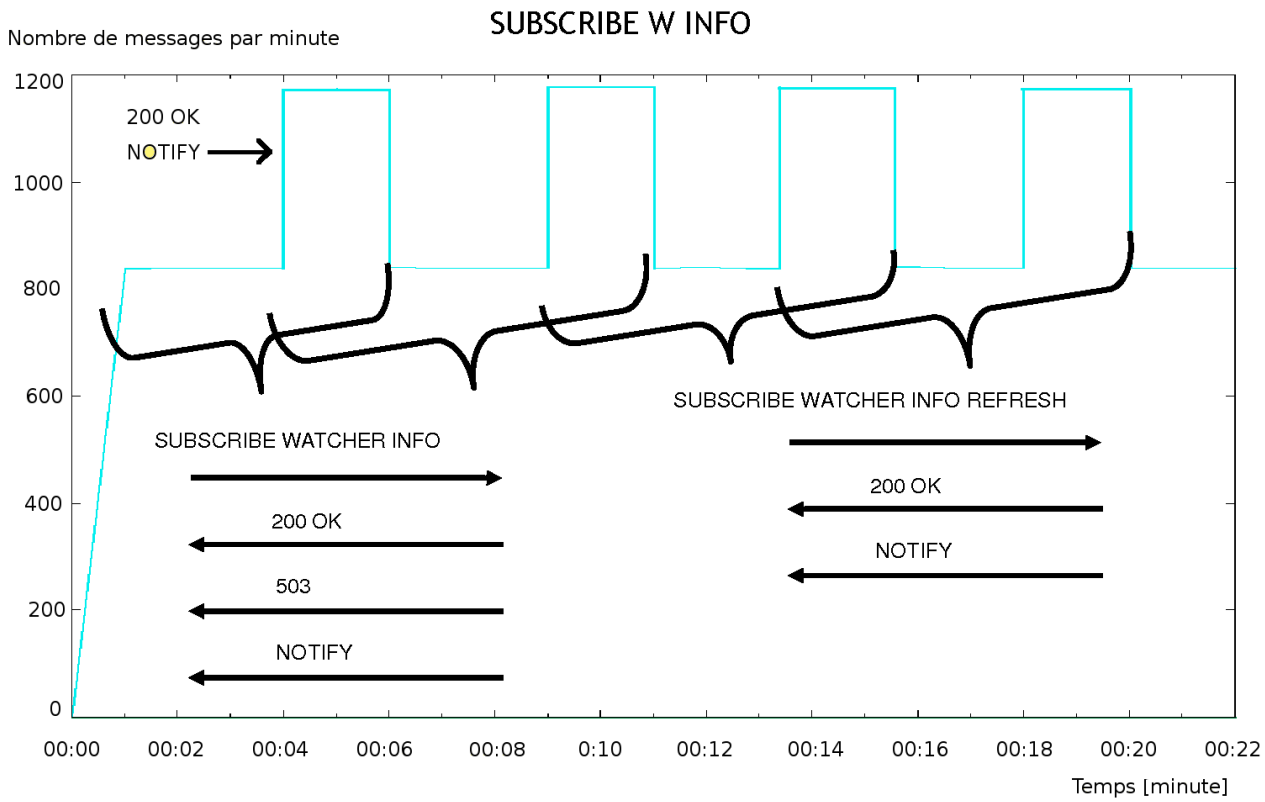


Figure 6.6 – L'injection des messages SUBSCRIBE WATCHER INFO à taux constant

6.3.2.2 Le trafic de stress

Le trafic de stress permet de tester trois cas de l'algorithme de contrôle de flux (figure 6.7). Le premier cas est la non-surcharge, aucun message ne doit être refusé si le système n'est pas exploité au maximum. Le deuxième cas teste la limite de la capacité du système, quelques refus sont observés. Le troisième cas teste la surcharge complète du système, il doit continuer à fonctionner avec de très bonnes capacités en acceptant autant de messages que sa capacité nominale le permet, en d'autres termes, sa capacité n'est pas affectée par une surcharge.

Ce trafic est en compétition avec le trafic de PUBLISH, car il n'est composé que de nouveaux messages ; ceux-ci, ainsi que le trafic de PUBLISH, sont tous refusables par le contrôle de flux de l'ASR.

6.4 L'impact de l'utilisation des sondes sur le contrôle d'admission

Les résultats montrent, par l'intermédiaire des figures 6.8 à 6.11, qu'il est nécessaire d'utiliser les sondes avec l'application Presence.

La figure 6.8 montre l'évolution des PUBLISH avec et sans l'utilisation des sondes. Comme attendu, certains PUBLISH sont refusés, pendant la surcharge du système, lorsque le système utilise les sondes. En ne les utilisant pas, les messages sont tous acceptés en provoquant une augmentation importante du temps de service, le client SiPp retransmet alors les PUBLISH, car les messages sont considérés perdus.

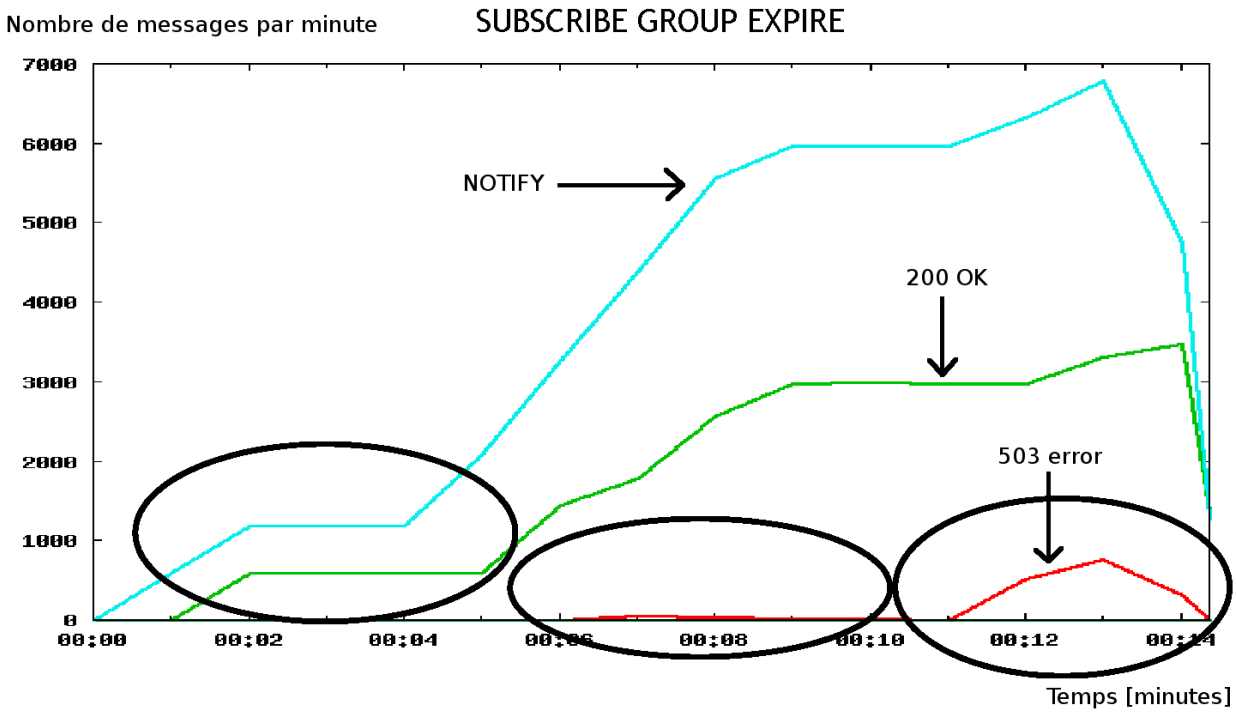


Figure 6.7 – L'injection des messages SUBSCRIBE GROUP à taux variable.

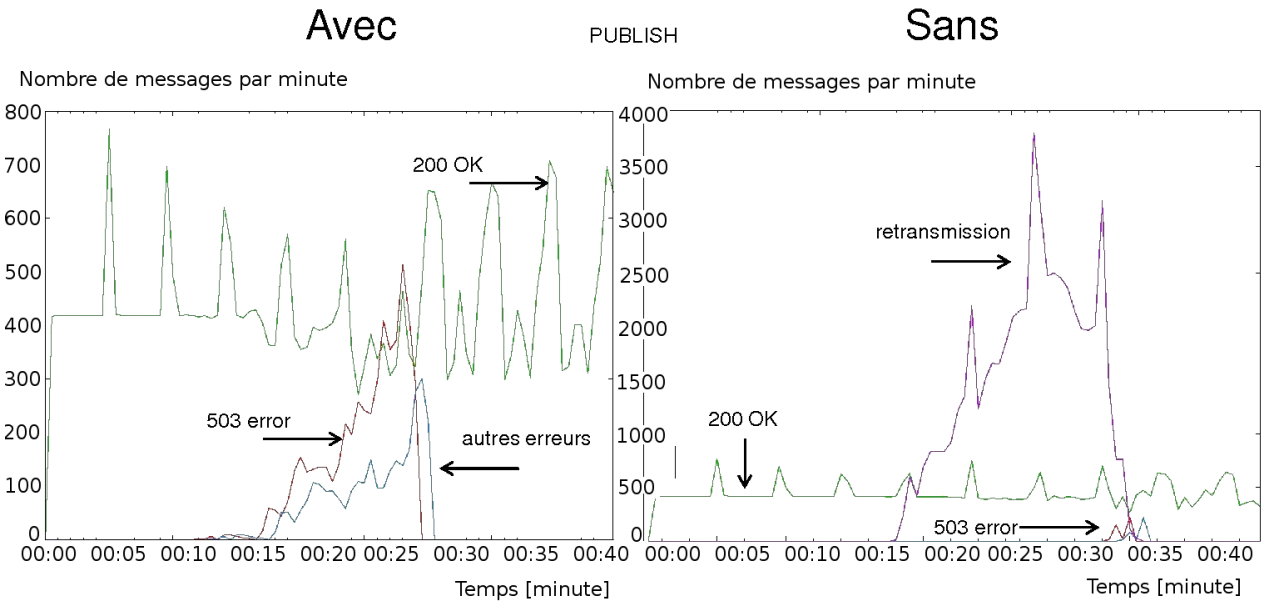


Figure 6.8 – L'injection des messages PUBLISH à taux constant.

Lors de la surcharge du système, un pic de NOTIFY est observé. Il est dû à l'expiration des documents de présence provoquée par des refus de PUBLISH (figure 6.9).

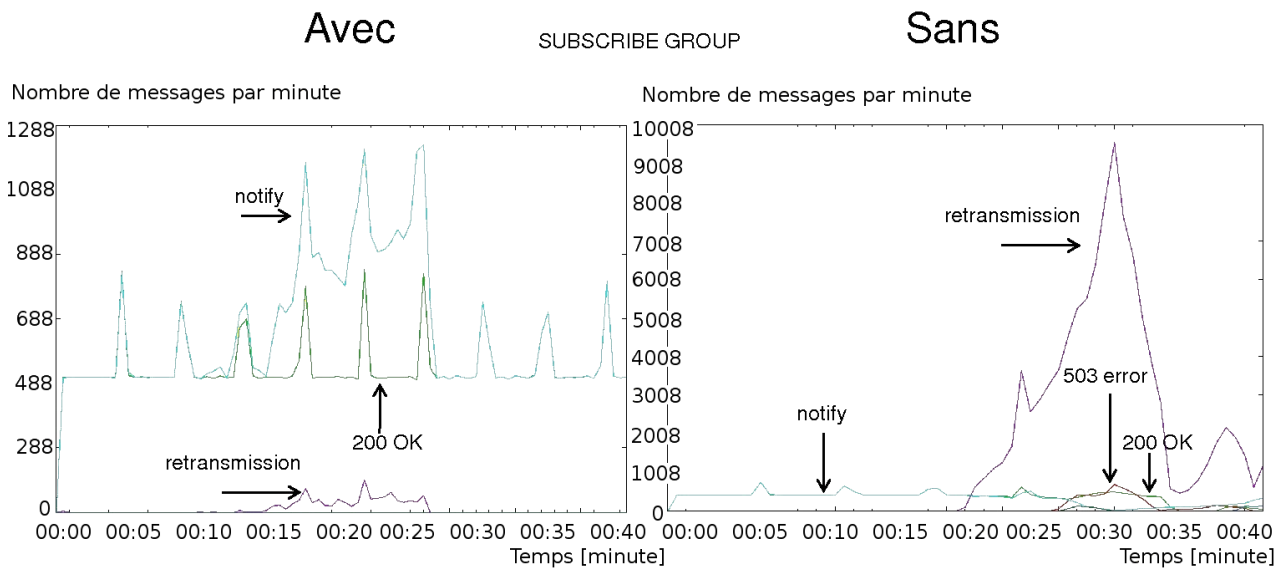


Figure 6.9 – L'injection des messages SUBSCRIBE GROUP à taux constant.

Avec les sondes, la gestion des SUBSCRIBE WATCHER INFO n'est pas impactée par la charge induite par le trafic de stress. Sans les sondes, les SUBSCRIBE WATCHER INFO sont retransmis induisant une augmentation des NOTIFY (figure 6.10).

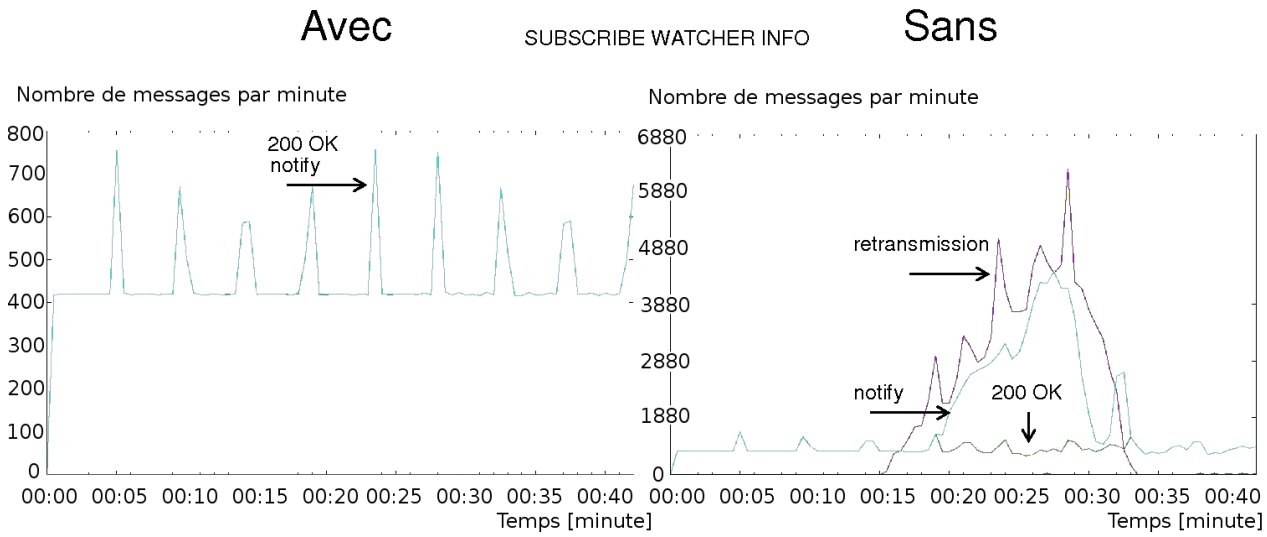


Figure 6.10 – L'injection des messages SUBSCRIBE WATCHER INFO à taux constant

Avec l'utilisation des sondes, les SUBSCRIBE GROUP du trafic de stress commencent à être refusés lorsque le système atteint sa capacité limite pour garder des temps de traitement courts (figure 6.12). Si les sondes ne sont pas utilisées, alors un nombre important de retransmissions est observé. Ces messages supplémentaires chargent le système jusqu'à ce que la charge devienne tellement importante, que le contrôle d'admission se déclenche en refusant des messages. La charge globale est sous-estimée.

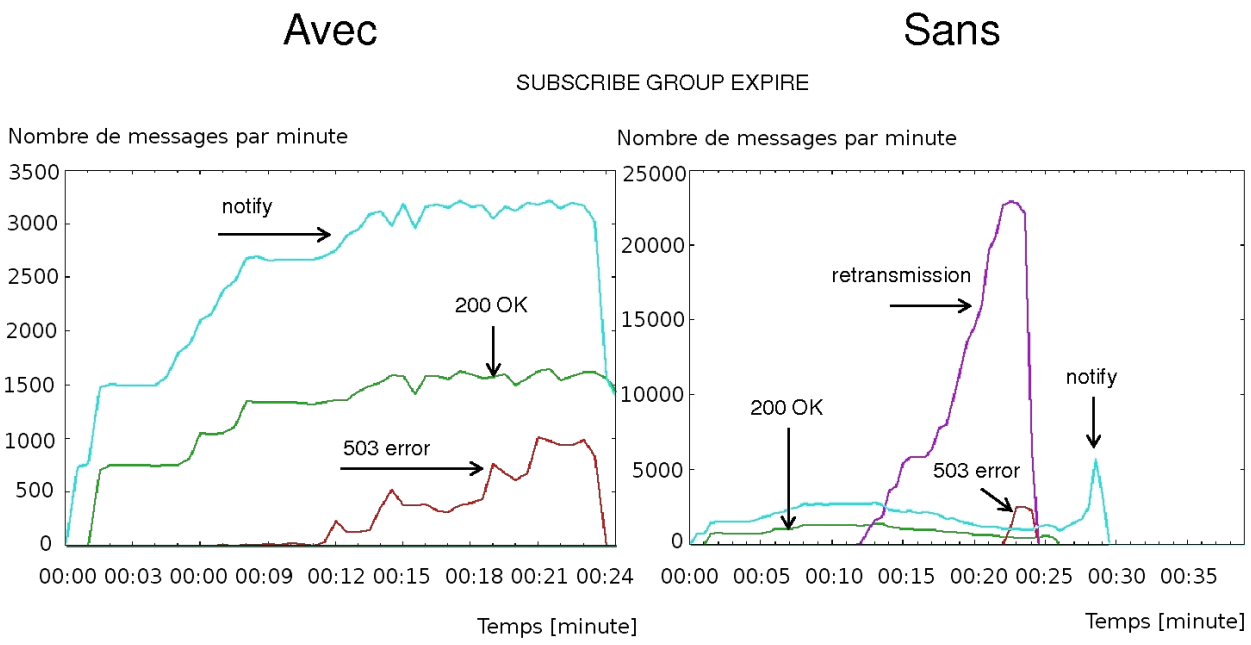


Figure 6.11 – L'injection des messages SUBSCRIBE GROUP à taux variable

Les temps de réponse deviennent extrêmement importants si les sondes ne sont pas utilisées. Les temps passent de 300 ms à trois secondes, c'est pour cette raison qu'il y a autant de retransmissions (figure 6.12).

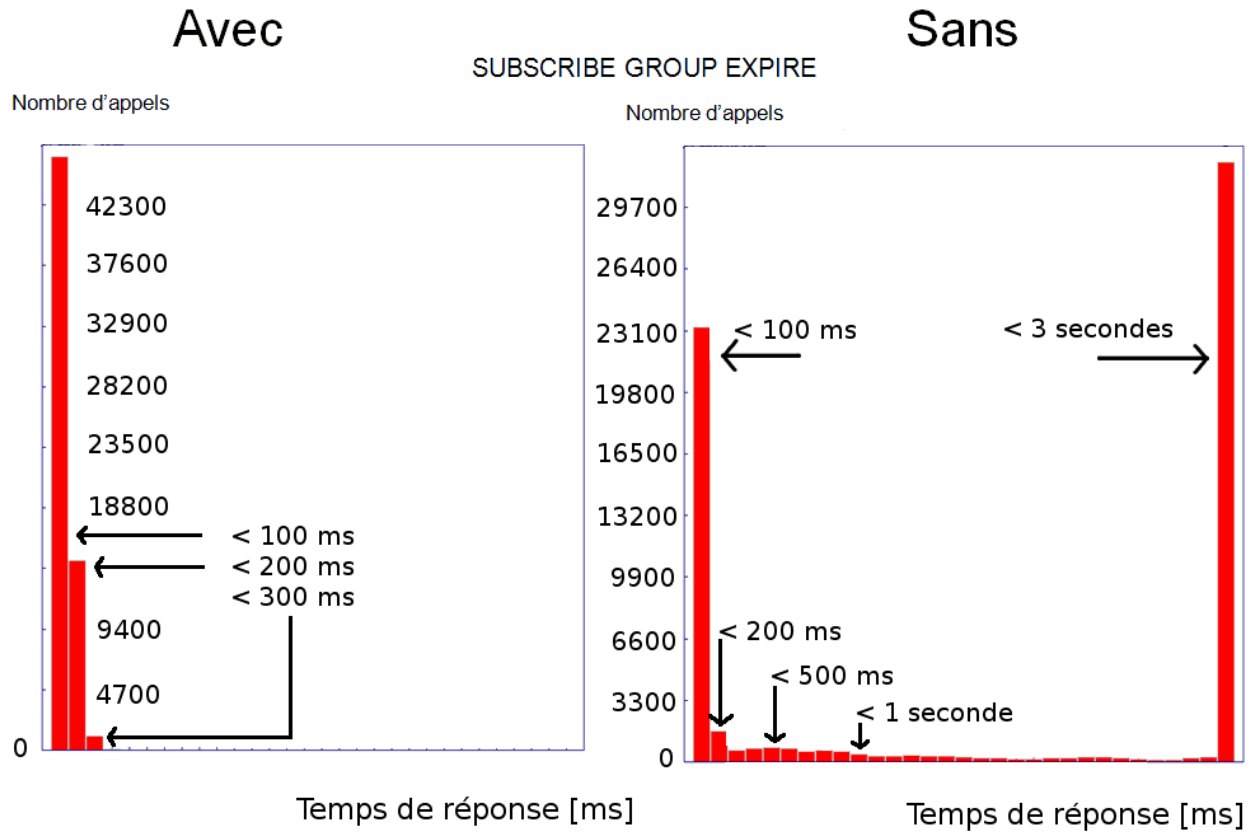


Figure 6.12 – Les temps de réponse du trafic de stress, les SUBSCRIBE GROUP

6.5 Le contrôle de charge

Le contrôle de charge reste opérationnel avec l'activation des sondes, car l'io_handler répartit équitablement les messages sur les six agents. La figure 6.13 montre que l'évolution des RTT en fonction du temps est presque identique. Les agents partagent équitablement la charge de traitement des messages.

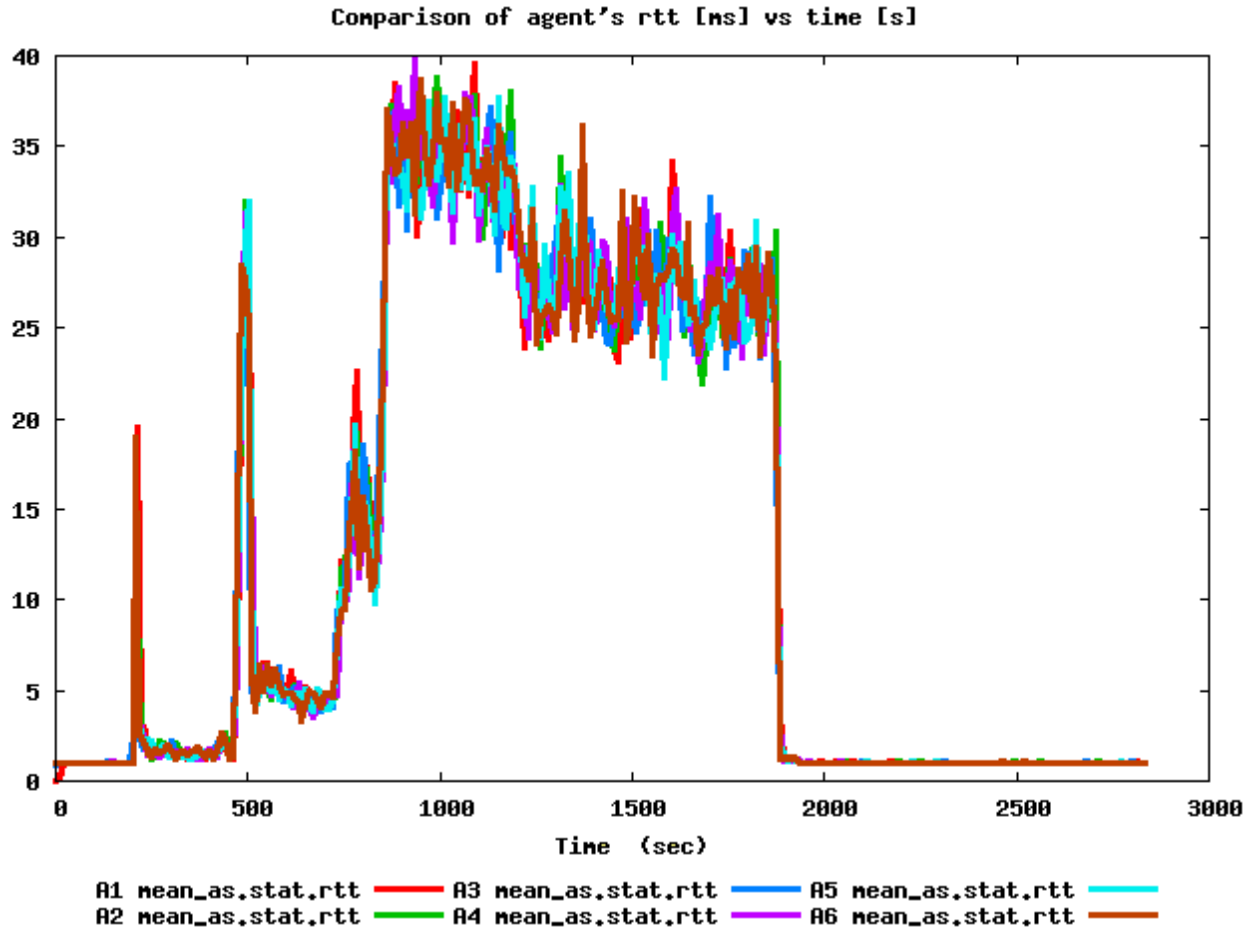


Figure 6.13 – Valeur moyenne des RTT en fonction du temps des six agents du groupe Presence Server. Les valeurs statistiques sont réalisées sur des intervalles de dix secondes.

6.6 Conclusion

L'étude de cas révèle que le contrôle d'admission est amélioré par l'utilisation de sondes du metering service. Trois cas sont testés, le premier est la non-surcharge, aucun message n'est refusé si le système n'est pas exploité au maximum. Le deuxième cas teste la limite de la capacité du système, quelques refus sont observés. Le troisième cas teste la surcharge complète du système, il continue à fonctionner avec de très bonnes capacités en acceptant autant de messages que sa capacité nominale le permet, sa capacité n'est pas affectée par une surcharge.

Lorsque les sondes sont désactivées, l'io_handler laisse passer trop de messages, dégradant en conséquence le temps de service. La charge globale du système est sous-estimée, l'utilisation des sondes permet d'augmenter le RTT pour que celui-ci prenne en compte les communications intergroupes en laissant intact le contrôle de partage de charge.

Chapitre 7

Conclusion et perspectives

Le contrôle d'admission de l'ASR d'Alcatel-Lucent est géré par deux algorithmes : le premier est l'algorithme du contrôle de flux et le deuxième est l'algorithme de l'évaluation de la charge globale du système.

Le premier algorithme a été analysé à travers deux modèles théoriques. Les prédictions de ces modèles sont identiques, excepté sur l'effet de la longueur de la fenêtre et sur le taux moyen de messages refusés. Cependant, ils ont montré que la longueur de la fenêtre possède une très grande influence sur le RTT et le temps de traitement des messages et qu'il est par conséquent important de maîtriser son évolution en simplifiant l'algorithme. Les modèles auraient nécessité une vérification très approfondie par une série d'expériences. Faute de temps et de moyens, nous avons présenté une étude de cas pour illustrer le bon comportement du système.

Le deuxième algorithme est nouveau et permet à un agent de prendre en considération ses voisins pour communiquer une information de charge à l'io_handler. L'étude sur le traitement des messages a montré qu'il était impossible de mesurer et/ou calculer le temps de traitement des messages sans faire appel à la couche applicative. Cette possibilité va à l'encontre de la ligne de conduite de l'équipe d'Alcatel-Lucent qui prône le développement de fonctionnalités génériques. Une approche statistique a alors été mise en place pour détecter rapidement le début d'une surcharge. Le RTT mesuré par une fonction de traitement des délais des opérations asynchrones représente la charge du système.

Enfin, un outil a été spécifiquement développé pour exploiter au mieux un service de monitoring de la plateforme. Intégré dans la plateforme, il a été très utile pour mener notre réflexion.

Bibliographie

[1] API of the Class javax.servlet.http.HttpServlet. Java Sun.

[2] API of the Interface java.lang.Runnable. Java Sun.

[3] API of the Interface javax.servlet.Servlet. Java Sun.

[4] Faq control flow. *Alcatel-Lucent*.

[5] Faq distributed session. *Alcatel-Lucent*.

[6] Faq metering service. *Alcatel-Lucent*.

[7] Man of function poll() . Linux manuel.

[8] The nononsensguide. *Alcatel-Lucent*.

[9] Alcatel-lucent 5350 application server runtime 3.4, operation and maintenance guide. *Alcatel-Lucent*, 2010.

[10] Joël Repiquet Ahmed Maouche, François Leygues. Alcatel-lucent 5350 application server runtime, sip proxy registrar. *Alcatel-Lucent*, 2009.

[11] Michael Franz Andreas Gal, Peter H. Fröhlich. An efficient execution model for dynamically reconfigurable component software. Technical report, Department of Information and Computer Science, University of California, Irvine, 2002.

[12] Eugene Charniak and Drew McDermott. *Introduction to Articial Intelligence*. Addison-Wesley Publishing Company, 1985.

[13] Joël Repiquet François Leygues, Thomas Froment. Telecommunication application servers, the alcatel-lucent java carrier-grade runtime. *Alcatel-Lucent*, 2007.

[14] Philippe Goujon. *Informatique et Rationalité, Approche épistémologique*. Faculté d'Informatique, Université de Namur, 2008.

[15] Jan Goyvaerts. Les Regex.

[16] Gilles Guerin. *La planète raison ou l'irrationnel aujourd'hui*. Al Editions, 2004.

[17] John Miller Dow Meiklejohn Immanuel Kant. *Critique of Pure Reason*. Dover Philosophical Classics Series, 2003.

[18] G. Camarillo A. Johnston J. Peterson R. Sparks M. Handley E. Schooler J. Rosenberg, H. Schulzrinne. Rfc 3261 on session initiation protocol, definition of session. *Internet RFC*, page 23, 2002.

[19] G. Camarillo A. Johnston J. Peterson R. Sparks M. Handley E. Schooler J. Rosenberg, H. Schulzrinne. Rfc 3261 on session initiation protocol, dialog. *Internet RFC*, page 69, 2002.

[20] G. Camarillo A. Johnston J. Peterson R. Sparks M. Handley E. Schooler J. Rosenberg, H. Schulzrinne. Rfc 3261 on session initiation protocol, retransmission. *Internet RFC*, page 124, 2002.

[21] G. Camarillo A. Johnston J. Peterson R. Sparks M. Handley E. Schooler J. Rosenberg, H. Schulzrinne. Rfc 3261 on session initiation protocol, transaction. *Internet RFC*, page 122, 2002.

[22] James F. Kurose and Keith W. Ross. *Computer Networking : A Top-Down Approach*. Addison-Wesley Publishing Company, USA, 5th edition, 2009.

[23] John Murphy Mircea Trofin. Removing redudant context management services in contextual composition frameworks. *Performance Engineering Laboratory, Dublin City University*, 2002.

[24] Karl Popper. *Le réalisme et la science*. Paris Edition Hermann, 1990.

[25] Irfan Pyarali, Tim Harrison, Douglas C. Schmidt, and Thomas D. Jordan. Proactor - an object behavioral pattern for demultiplexing and dispatching handlers for asynchronous events. Technical report, Washington University, 1997.

[26] Olivier Jacques Richard Gayraud. Sipp reference documentation. Technical report, 2004.

[27] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2 : Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[28] Douglas C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Commun. ACM*, 38(10) :65–74, 1995.

[29] Dimitri Tombroff. 5350 distributed session management. Technical report, Alcatel-Lucent.

[30] C. Shen A. Abdelal V. Hilt, E. Noel. Design considerations for session initiation protocol (sip) overload control. *Internet IETF Draft, work in progress*, page 16, January 12 2010.

[31] J. Virtamo. Flow control : window mechanism, April 2007.

Chapitre 8

Annexes

8.1 L'impact de la longueur de la fenêtre sur le RTT et le temps de traitement des messages

L'évolution du RTT, des valeurs des sondes du metering service et du temps de traitement des messages est linéaire par rapport à la longueur de la fenêtre (voir la figure 8.1). L'arrivée des messages est à taux moyen constant plus ou moins trois fois plus important que la capacité de la plateforme, les temps d'interarrivées sont constants et les messages sont des SUBSCRIBE GROUP.

8.2 Le metering service

Les sections suivantes présentent une liste non-exhaustive des informations monitorées du réacteur, du thread pool et de l'hôte.

8.2.1 Le monitoring du réacteur

- as.stat.reactor.ReactorName.task.elapsed : moyenne du temps (en ms) consommé par l'exécution de tâches programmées par un timer.
- as.stat.reactor.ReactorName.task.queueSize : longueur moyenne de la file des tâches programmées par un timer.
- as.stat.reactor.ReactorName.task.rate : moyenne du nombre de tâches programmées par un timer par seconde.
- as.stat.reactor.ReactorName.timer.elapsed : temps moyen consommé par le réacteur des timers (en ms)
- as.stat.reactor.ReactorName.timer.active : nombre moyen de timers actifs pour un réacteur donné dont le nom est ReactorName.
- as.stat.reactor.ReactorName.timer.gcCount : nombre moyen de timers annulés, c'est-à-dire retirés de la file des timers.
- as.stat.reactor.ReactorName.timer.gcElapsed : temps moyen consommé par le garbage collector des timers. timer *garbage collector*.
- as.stat.reactor.ReactorName.socket.bytesPerWrite : nombre moyen de bytes envoyés via l'appel système write().
- as.stat.reactor.ReactorName.socket.bytesPerRead : nombre moyen de bytes reçu via l'appel système read().
- as.stat.reactor.ReactorName.socket.outputRate : nombre moyen de bytes envoyés par seconde.
- as.stat.reactor.ReactorName.socket.inputRate : nombre moyen de bytes reçus par seconde.
- as.stat.reactor.ReactorName.socket.elapsedPerRead : moyenne des temps consommés pour lire les données des sockets.
- as.stat.reactor.ReactorName.socket.buffered : nombre de bytes non envoyés pour cause de socket rempli.

Voici ci-dessous un exemple avec le réacteur HTTP.

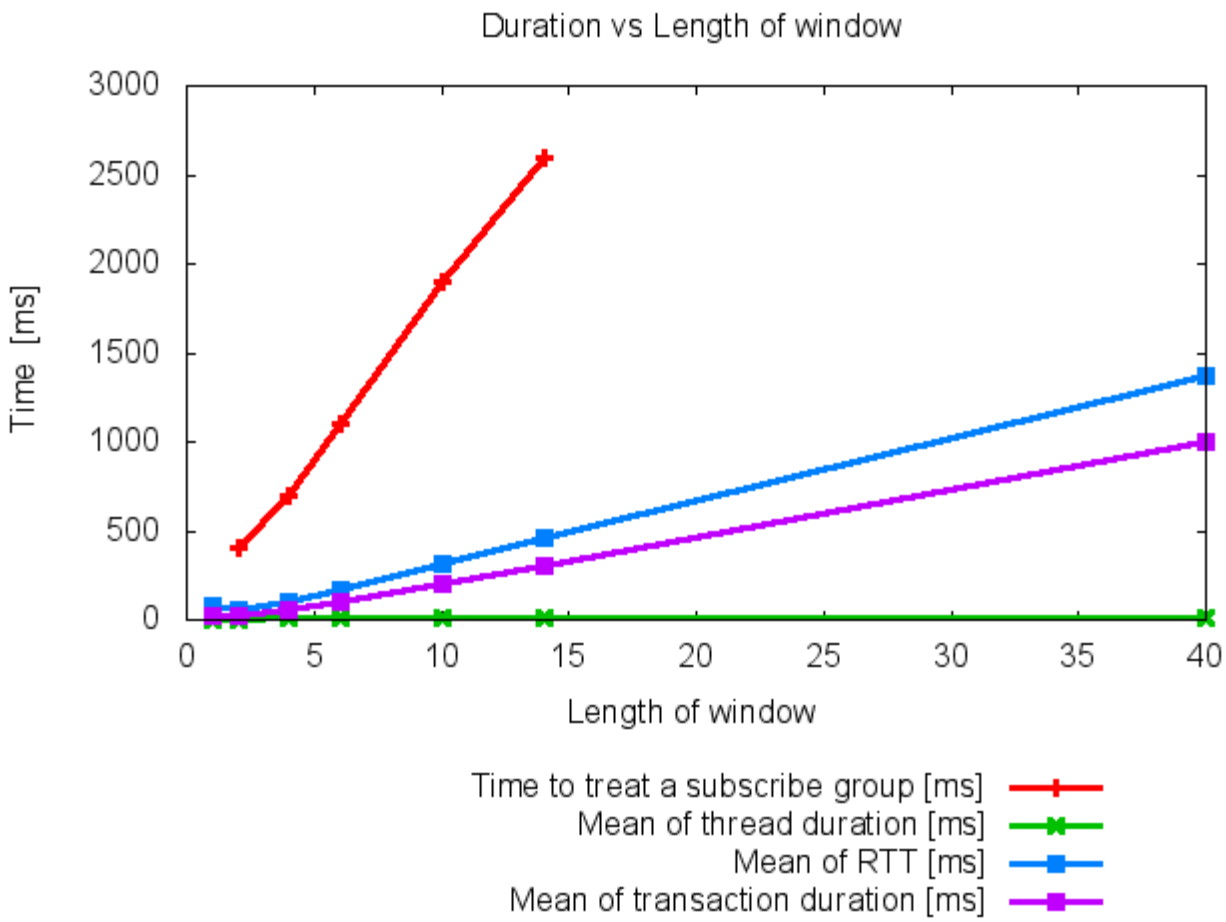


Figure 8.1 – L'évolution du RTT, des valeurs des sondes du metering service et du temps de traitement des messages est linéaire par rapport à la longueur de la fenêtre.

```
2009-09-21 23:38:50,784 Metering INFO  as.stat.reactor.Http.socket.inputRate  - [Rate] Value=2385657/53343886; Mean
    =238565.700/33360.779; Deviation=288327.999/135576.274; Min=0/0; Max=623082/671147
2009-09-21 23:38:50,784 Metering INFO  as.stat.reactor.Http.task.queueSize  - [Counter] Value=0/0; Mean=0/0; Deviation=0/0; Min
    =0/0; Max=0/0
2009-09-21 23:38:50,785 Metering INFO  as.stat.reactor.Http.task.elapsed  - [Counter] Value=0/0; Mean=0/0; Deviation=0/0; Min=0/0;
    Max=0/0
2009-09-21 23:38:50,785 Metering INFO  as.stat.reactor.Http.timer.elapsed  - [Counter] Value=0/9; Mean=0/0.040; Deviation=0/0.302;
    Min=0/0; Max=0/4
2009-09-21 23:38:50,785 Metering INFO  as.stat.reactor.Http.socket.buffered  - [Gauge] Value=0/0; Mean=0/0; Deviation=0/0; Min=0/0;
     Max=0/0
2009-09-21 23:38:50,785 Metering INFO  as.stat.reactor.Http.socket.bytesPerRead  - [Counter] Value=2822370/53663055; Mean
    =4049.311/5975.841; Deviation=6779.446/7794.245; Min=24/8; Max=32768/32768
2009-09-21 23:38:50,786 Metering INFO  as.stat.reactor.Http.socket.outputRate  - [Rate] Value=1668243/36789588; Mean
    =166824.300/23007.872; Deviation=200323.151/93477.004; Min=24/0; Max=431187/460654
2009-09-21 23:38:50,786 Metering INFO  as.stat.reactor.Http.timer.active  - [Gauge] Value=1/1; Mean=0.984/1; Deviation=0.127/0; Min
    =1/1; Max=1/1
2009-09-21 23:38:50,787 Metering INFO  as.stat.reactor.Http.socket.bytesPerWrite  - [Counter] Value=1980762/37102107; Mean
    =2071.927/2306.915; Deviation=1787.878/1762.972; Min=4/4; Max=4096/4096
2009-09-21 23:38:50,787 Metering INFO  as.stat.reactor.Http.socket.elaspedPerRead  - [Counter] Value=3354/54952; Mean=4.861/6.156;
    Deviation=9.184/9.408; Min=0/0; Max=71/119
2009-09-21 23:38:50,787 Metering INFO  as.stat.reactor.Http.task.rate  - [Rate] Value=0/0; Mean=0/0; Deviation=0/0; Min=0/0; Max
    =0/0
```

8.2.2 Le monitoring du thread pool

– as.stat.tpool.elapsed : temps moyen (ms) consommé par les threads en exécution.
– as.stat.tpool.queue : longueur moyenne de la file de tâchess programmées. Cette file est utilisée pour accumuler des tâches quand aucun thread n'est disponible.
– as.stat.tpool.working : nombre moyen de threads en exécution.
– as.stat.tpool.idle : nomble moyen de threads libres, c'est-à-dire prêt à recevoir une tâche. [1]

Voici ci-dessous un exemple du suivi du thread pool.

1. Les emplacements dans le thread pool ne sont pas forcément occupés par un thread, ils sont vides. Quand une tâche doit être traitée, un thread est créé dans un emplacement. Ce thread reste actif pour traiter la tâche suivante, si elle n'arrive pas après quelques secondes, alors le thread est détruit et l'emplacement du thread pool est vide.

```
2009-09-21 23:41:00,868 Metering INFO  as.stat.tpool.idle  - [Gauge] Value=60/60; Mean=30.909/40.045; Deviation=20.248/38.486; Min
    =0/0; Max=61/100
2009-09-21 23:41:00,868 Metering INFO  as.stat.tpool.elapsed  - [Counter] Value=2581/156151; Mean=0.153/0.230; Deviation
    =1.135/1.567; Min=0/0; Max=60/94
2009-09-21 23:41:00,868 Metering INFO  as.stat.tpool.queue  - [Gauge] Value=1/1; Mean=1.405/1.185; Deviation=4.467/5.703; Min=0/0;
    Max=61/142
2009-09-21 23:41:00,869 Metering INFO  as.stat.tpool.working  - [Gauge] Value=1/1; Mean=0.801/0.583; Deviation=2.126/3.016; Min
    =0/0; Max=43/100
```

8.2.3 Le monitoring de l'hôte

– as.stat.memory.usedkb : quantité moyenne de mémoire utilisée en kb.
– as.stat.memory.freekb : quantité moyenne de mémoire libre en kb.
– as.stat.cpu.system : moyenne du *system cpu*
– as.stat.cpu.user : moyenne du *user cpu*
– as.stat.cpu.idle : moyenne du *idle cpu*

Voici ci-dessous un exemple de suivi d'un hôte.

```
2009-09-21 23:42:00,899 Metering INFO  as.stat.cpu.user - [Counter] Value=107/288238; Mean=5.350/80.558; Deviation=7.220/3685.921;
    Min=0/0; Max=22/220524
2009-09-21 23:42:00,899 Metering INFO  as.stat.cpu.system - [Counter] Value=35/74070; Mean=1.750/20.702; Deviation=1.728/810.234;
    Min=0/0; Max=8/48476
2009-09-21 23:42:00,900 Metering INFO  as.stat.memory.usedkb - [Gauge] Value=57805/57805; Mean=57238.794/35687; Deviation
    =5692.885/31303.987; Min=57805/5053; Max=57805/143064
2009-09-21 23:42:00,900 Metering INFO  as.stat.memory.freekb - [Gauge] Value=194098/194098; Mean=192216.002/216216.088; Deviation
    =19019.729/31304.086; Min=194098/108839; Max=194098/246850
2009-09-21 23:42:00,900 Metering INFO  as.stat.cpu.idle - [Counter] Value=1885/2061040; Mean=94.250/576.031; Deviation
    =10.839/29949.945; Min=72/0; Max=113/1791822
```

8.3 Meter2gnuplot

L'outil meter2gnuplot est un support graphique au metering service et un outil de corrélation, car il permet de croiser et d'illustrer des informations statistiques de monitoring de la plateforme. Il est possible de corréler ces données avec des événements très fins dans les agents et les io_handler comme la longueur des files, le pourcentage de retransmission, les temps de traitement, etc.).

Cet outil facilite la compréhension de l'effet d'une surcharge sur le système. Il a été principalement utilisé pour étudier le comportement du thread pool (voir la section 2.2.2), des transactions distribuées (voir la section 2.2.3) et des fonctions de calcul du RTT (voir la section 4.3.4).

Meter2gnuplot a connu deux évolutions. La première version était un prototype, mais l'ajout de nouvelles fonctions le rendait de plus en plus complexe. La deuxième version est maintenable et autorise l'ajout de nombreuses fonctionnalités.

La première partie illustre l'utilisation de Meter2gnuplot et la deuxième partie explique le fonctionnement de l'outil.

8.3.1 L'utilisation de Meter2gnuplot

Meter2gnuplot est un outil en ligne de commandes prenant un *input* et produisant un *output* (figure 8.2). L'application peut recevoir les données de monitoring sur l'entrée standard *stdin* grâce à la commande *tail -f* ou *cat* qui peut être couplée à la commande *ssh* pour atteindre des hôtes distants. Une option permet de se passer de l'entrée standard en prenant en entrée le chemin du fichier log contenant les données.

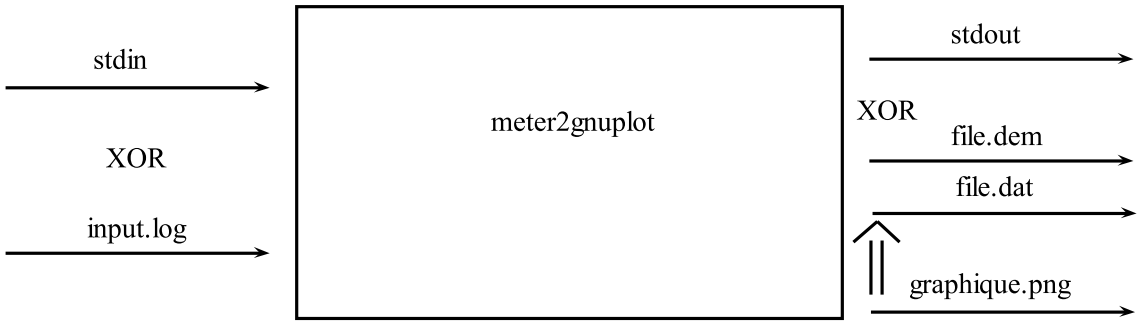


Figure 8.2 – Les entrées sorties de Meter2gnuplot.

Les sorties de Meter2gnuplot sont d'une part la sortie standard *stdout* (figure 8.3) et d'autre part un fichier dem et dat ou png (figure 8.4). La sortie *stdout* autorise une petite application, nommée rtGnuplot, de tracer en temps réel un graphique avec Gnuplot.

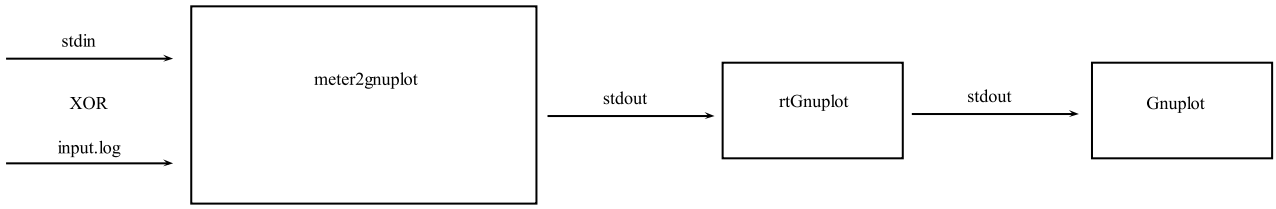


Figure 8.3 – Meter2gnuplot associé à rtGnuplot pour tracer des graphiques avec une mise à jour automatique en temps réel.

L'utilisation des fichiers dem et dat permet à Gnuplot, quand l'utilisateur le demande, de tracer un graphique. Le premier fichier dem possède la description des données telles que le nom des courbes, des axes, le titre du graphique, etc., et le deuxième fichier possède une matrice de données à tracer.

Les options présentées dans le tableau 8.5 peuvent être combinées à volonté. Les éléments de l'ASR à monitorer sont

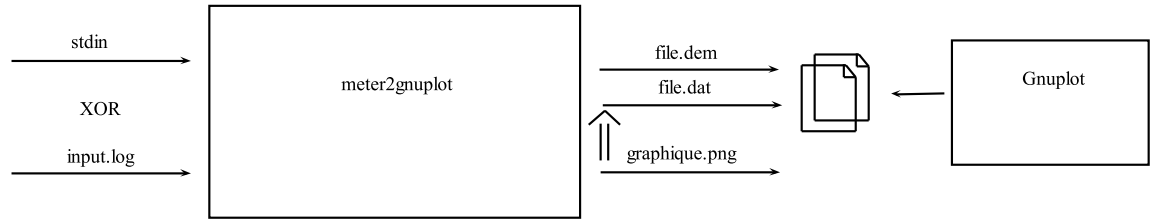


Figure 8.4 – Meter2gnuplot génère les fichiers .dem et .dat pour que Gnuplot trace les courbes, la mise à jour du graphique est possible en temps réel sur l'initiative de l'utilisateur.

sélectionnés grâce à une liste de nom de logger : <metering names>. Le metering service fournit des informations statistiques, calculées toutes les dix secondes, [2] qui peuvent être tracée sur un graphique grâce aux options –m pour afficher la valeur minimale, –M afficher la valeur maximale, –a pour afficher la valeur moyenne, –d pour afficher l'écart-type. Ces quatre options ont la particularité de pouvoir se combiner, par exemple –mM <metering names> affiche à la fois le minimum et le maximum de la liste des metering names en paramètres, –ad affiche la moyenne et l'écart type, –admM affiche la moyenne, l'écart type, la minimum et le maximum.

La succession de noms de logger <metering names> peut être exprimée grâce aux Regex (voir [15]) comme le montre l'exemple de la figure 8.6, où l'évolution de la moyenne, du maximum, du minimum et de l'écart-type du RTT et la moyenne de toutes les durées est tracée (figure 8.7). La commande *ssh hôteDistant tail -f chemin/fichier* | permet de diriger en temps réel le contenu du fichier de log vers l'entrée standard de Meter2gnuplot.

L'option –png permet de générer, en fin de test, [3] le graphique présenté à la figure 8.7.

8.4 Le fonctionnement de Meter2gnuplot

Meter2gnuplot est un outil de conversion de format en temps réel. La transformation et les difficultés à prendre en compte sont présentées avant de détailler la solution.

8.4.0.1 La transformation de format

Le format du metering service est défini dans cette section, mais est abordé dans la section 2.4 et des exemples sont placés dans les annexes (voir la section 8.2). Dans un souci de concision, la présentation du format de gnuplot se limite aux possibilités qu'offre Meter2gnuplot.

Le format du metering service Le format du metering service est défini dans le cadre de la figure 8.8. Les valeurs situées à gauche de la barre de fraction sont des statistiques calculées pendant un intervalle de dix secondes [4] et celles situées à droite de la barre sont les statistiques accumulées depuis la définition du logger (voir la section 2.3). Les lignes associées à leur logger sont écrites toutes les dix secondes par bloc entier dans le fichier de log.

Le format de Gnuplot Composé de deux fichiers, le premier est la description des données (voir la figure 8.9) et le deuxième fichier contient les données représentées par une matrice de valeurs.

8.4.0.2 Les difficultés

Les trois grandes difficultés sont la tolérance aux fautes, la nécessité de convertir le format des données sans les accumuler en mémoire et la multiplicité des options de conversion. Les fautes comme des exceptions ou des crashs peuvent modifier

2. Ce temps est paramétrable.
3. En tuant le processus grâce au signal CTRL-C
4. Ce temps est paramétrable.

nom de l'option	explication
-o ou –output <fileName>	La sortie sera un fichier fileName.dem .dat .png. Sinon elle est une concaténation de la représentation des données .dem et de la matrice de données .dat vers la sortie standard stdout.
-i ou –input <fileNames.log>	Le fichier d'entrée. Mais ne fonctionne pas avec ssh et désactive stdin.
-t ou –title <titleOfTheGraph>	Le titre qui apparaît sur le graphique.
-p ou –png <>	Permet la génération d'un png du graphique, le nom du fichier est fileName.png
-h ou –help <>	Affiche l'aide
-s ou –startPlotDate <year :month :day hour :min :sec>	Heure de début pour commencer à tracer le graphique. Un exemple pour le format : "2009 :12 :18 13 :34 :00"
-e ou –endPlotDate <year :month :day hour :min :sec>	Heure de début pour commencer à tracer le graphique. Un exemple pour le format : "2009 :15 :00 19 :30 :00"
–m <metering names>	Affiche la courbe représentant la valeur minimale pendant l'intervalle de mesure pour chaque nom de metering.
–M <metering names>	Affiche la courbe représentant la valeur maximale pendant l'intervalle de mesure pour chaque nom de metering.
–a <metering names>	Affiche la courbe représentant la valeur statistique moyenne pour chaque nom de metering.
–d <metering names>	Affiche l'écart type pour chaque nom de metering.

Figure 8.5 – Les options graphiques et de sélection de données à tracer de Meter2gnuplot.

```
ssh nxuser@nx0026 tail -f  ...
/opt/proxy/var/log/AliasPresenceServerGroup__callout_agent_1/msg.log | ...
./meter2gnuplot  -admM as.stat.rtt -a as.stat.*duration.*  ...
--output outputDuration --title "RTT and .*duration.*" -adM as.stat.*elapsed.* --png
;gnuplot -persist outputDuration.dem
```

Figure 8.6 – Un exemple de ligne de commande pour utiliser Meter2gnuplot dans un environnement distribué.

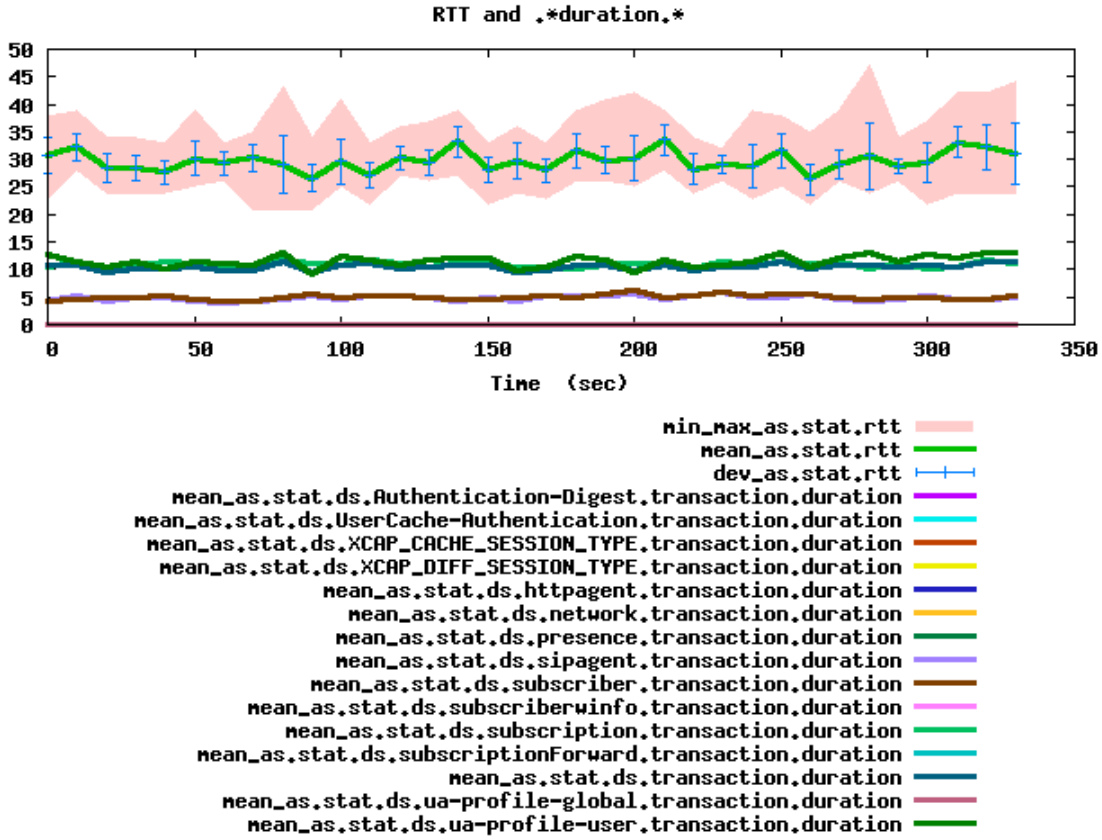


Figure 8.7 – Le graphique issu de la commande présentée à la figure 8.6. L'évolution de la moyenne, du maximum, du minimum et de l'écart-type du RTT et la moyenne de toutes les durées sont tracées.

```
année-mois-jour h:m:s,ms Metering Log4jLevel  meteringName  - [Rate] Value=int/int;
Mean=float/float; Deviation=float/float; Min=float/float; Max=float/float

Exemple :

2009-09-21 23:38:50,784 Metering INFO  as.stat.reactor.Http.socket.inputRate  - [Rate]
Value=2385657/53343886; Mean=238565.700/33360.779; Deviation=288327.999/135576.274;
Min=0/0; Max=623082/671147
```

Figure 8.8 – Le format des données du metering service dans le fichier de log.

```
set title "RTT and .*duration.*"
set style data lines
set xlabel "Time  (sec)"
set key below
set style fill solid 0.2 noborder
plot"demo.dat" u 1:4:5 title "min_max_as.stat.rtt" w filledcurves ,
"demo.dat" u 1:2 title "mean_as.stat.rtt" w l lw 3,
"demo.dat" u 1:2:3 title "dev_as.stat.rtt"w errorbars lw 1,
"demo.dat" u 1:6 title "mean_as.stat.ds.Authentication-Digest.transaction.duration"
                ... w l lw 3,
"demo.dat" u 1:7 title "mean_as.stat.ds.UserCache-Authentication.transaction.duration"
                ... w l lw 3,
"demo.dat" u 1:8 title "mean_as.stat.ds.XCAP_CACHE_SESSION_TYPE.transaction.duration"
                ... w l lw 3,
"demo.dat" u 1:9 title "mean_as.stat.ds.XCAP_DIFF_SESSION_TYPE.transaction.duration"
                ... w l lw 3,
"demo.dat" u 1:10 title "mean_as.stat.ds.httpagent.transaction.duration" w l lw 3,
"demo.dat" u 1:11 title "mean_as.stat.ds.network.transaction.duration" w l lw 3,
"demo.dat" u 1:12 title "mean_as.stat.ds.presence.transaction.duration" w l lw 3,
"demo.dat" u 1:13 title "mean_as.stat.ds.sipagent.transaction.duration" w l lw 3,
"demo.dat" u 1:14 title "mean_as.stat.ds.subscriber.transaction.duration" w l lw 3,
"demo.dat" u 1:15 title "mean_as.stat.ds.subscriberwinfo.transaction.duration"
                ... w l lw 3,
"demo.dat" u 1:16 title "mean_as.stat.ds.subscription.transaction.duration" w l lw 3,
"demo.dat" u 1:17 title "mean_as.stat.ds.subscriptionForward.transaction.duration"
                ... w l lw 3,
"demo.dat" u 1:18 title "mean_as.stat.ds.transaction.duration" w l lw 3,
"demo.dat" u 1:19 title "mean_as.stat.ds.ua-profile-global.transaction.duration"
                ... w l lw 3,
"demo.dat" u 1:20 title "mean_as.stat.ds.ua-profile-user.transaction.duration" w l lw 3
```

Figure 8.9 – Le fichier de description de données .dem du graphique de la figure 8.7.

ou corrompre le fichier de log. Il est alors incomplet ou possède des informations supplémentaires inutiles pour monitorer l'ASR.

La multiplicité des options de conversion complique la génération du fichier de description des données .dem, car il est important d'assurer la correspondance entre la description de données et les colonnes pertinentes du fichier de données .dat.

8.4.0.3 La première solution

La première solution se basait sur l'utilisation d'un tableau statique où étaient accumulées les valeurs selon un ordre très précis. Les moyennes étaient rangées entre la colonne 0 et le nombre de loggerNames-1, et ainsi de suite. Le fichier de description de données était généré en conséquence.

Cette situation était impossible à gérer du fait du nombre trop élevé des combinaisons possibles entre les options de sélection des données (min, max, moyenne, écart-type). Le code devenait une succession interminable de if then else, les structures de données n'étaient pas flexible et l'ajout d'une nouvelle option compromettait toute l'application.

8.4.0.4 La deuxième solution

La deuxième solution consiste à exprimer toutes les contraintes dans un arbre syntaxique de la même manière qu'un compilateur. La non-accumulation et la correspondance entre les données et leur description sont assurées par cet arbre, car les données le garnissent comme des guirlandes sur un sapin.

Le principe Le principe consiste retenir toutes les informations et les contraintes dans un arbre syntaxique. La structure du noeud est unique et elle est présentée sur la figure 8.10. Une première version de l'arbre est construite à partir de l'analyse syntaxique des options présentées dans le tableau 8.5.

L'arbre syntaxique La structure de l'arbre : Le noeud racine est un noeud de type TREE_OPT_ROOT et possède deux fils. Contrairement à celui de droite de type TREE_CURVE, celui de gauche, de type TREE_OPT_COMMON, décrit toutes les options non relatives aux tracés des courbes. Le noeud de type TREE_OPT_COMMON caractérise les options title, output, input, png, help, startPlotDate et endPlotDate et leurs arguments sont stockés dans des noeuds de type TREE_OPT_COMMON_ARGS. Le noeud de type TREE_CURVE introduit un sous-arbre possédant toutes les informations relatives au tracé d'un metering name sur le graphique, ces options sont contenues dans des noeuds de type TREE_OPT_CURVE (m, M, a, d).

Les champs ival, fval et sval sont utilisés pour stocker des informations numériques ou des chaînes de caractères et le champ columnId est utilisé pour numéroter les noeuds de type TREE_OPT_CURVE à partir de la racine, ces valeurs numériques sont utilisées pour numéroter les étiquettes des colonnes dans le format de gnuplot (voir la section 8.4.0.1).

La construction de l'arbre : L'arbre est construit à partir de l'analyse syntaxique des options en deux étapes si les Regex sont utilisés pour sélectionner un ensemble de metering name (par exemple as.stat.*duration.*), sinon en une seule étape.

La figure 8.11 montre l'arbre relatif à l'exécution de la commande présentée à la figure 8.6, L'utilisation des Regex oblige le remplacement des noeuds de type TREE_CURVE par les metering names correspondant dans le fichier de log. Cette opération nécessite l'analyse syntaxique du début de ce fichier.

La figure 8.12 illustre l'arbre (beaucoup plus grand) après avoir remplacé tous les noeuds TREE_CURVE représentant un Regex. Pour des raisons de lisibilité, une partie de l'arbre est cachée.

Les vérifications La structure de l'arbre et les options sont vérifiées et une liste est présentée ci-dessous.

1. le fichier de log doit exister (–input <fichier.log>),
2. les options qui n'existent pas sont refusées,
3. le nombre d'arguments des options est vérifié (par exemple, le présence d'un titre de l'option –title <titre> est obligatoire),

```
typedef struct option_t * option_PTR;

struct optnode {
  int typeNode;
  int ival;
  int columnId;
  float fval;
  char* sval;
  struct optnode * left;
  struct optnode * right;
  struct optnode * father;
};

typedef struct optnode * OPTTREE;
typedef struct optnode   OPTNODE;
```

Figure 8.10 – La définition récursive de l'arbre syntaxique manipulé Meter2gnuplot.

4. au moins un metering name doit exister dans le fichier de log, les metering names doivent être identifiant (ils ne sont exprimés qu'une seule fois dans les options et la transformation des expressions régulières ne crée pas de doublon),
5. les expressions régulières doivent être bien formées,
6. seules les options de sélection de données (moyenne, écart-type, minimum et maximum) peuvent être combinées entre-elles.

La génération en temps réel du fichier de données .dat et le fichier de description .dem La génération du fichier de description de données .dem : La génération est possible à partir de l'arbre syntaxique, les noeuds sont parcourus dans le sens croissant du champ columnId. À chaque noeud de l'arbre correspond une règle de transformation.

1. Le noeud racine, de type TREE_OPT_ROOT, permet de générer les chaînes de caractères "*set title <titre>*" et "*set style data lines ; xlabel 'Time (sec)' ; set key below ; set style fill solid 0.2 noborder*".
2. Aucune règle de transformation n'est associée au noeud de type TREE_OPT_COMMON, car ce ne sont que des options sans rapport avec le tracé des courbes du graphique.
3. Les noeuds de type TREE_OPT_CURVE ne sont pas parcourus lors de l'unique lecture de l'arbre, mais sont atteints à partir du noeud de type TREE_CURVE.
4. Le noeud de type TREE_CURVE permet de rechercher les noeuds de type TREE_OPT_CURVE. Le premier noeud recherché possède l'information de moyenne (–a), la deuxième le minimum (–m), la troisième le maximum (–M) et la dernière l'écart type (–d). La recherche par type de noeud ou d'information est importante, car la forme de l'arbre syntaxique dépend de l'ordre des options de la ligne de commande de Meter2gnuplot. La méthode responsable la génération des lignes est placée en annexes dans la section 8.5.

La génération du fichier de données .dat : La génération du fichier de données .dat est réalisé successivement par ligne entière. Les blocs de metering names sont parsés dans le fichier de log et pour chacun d'eux, toutes les données sont parsées et placées dans l'arbre si le noeud correspondant existe. À la fin de l'analyse du bloc, l'arbre est parcouru dans le sens de lecture donnée par le champ columnId, les valeurs sont recopiées dans le fichier de données et les valeurs de l'arbre sont réinitialisées. Si une valeur est absente de l'arbre, un tiret est placé dans le fichier pour le signaler à gnuplot.

8.4.1 Conclusion

Meter2gnuplot est un outil pratique pour observer en temps réel et graphiquement l'évolution des sondes du metering service de l'ASR. Il a facilité la compréhension de l'effet d'une surcharge sur le système principalement en étudiant le

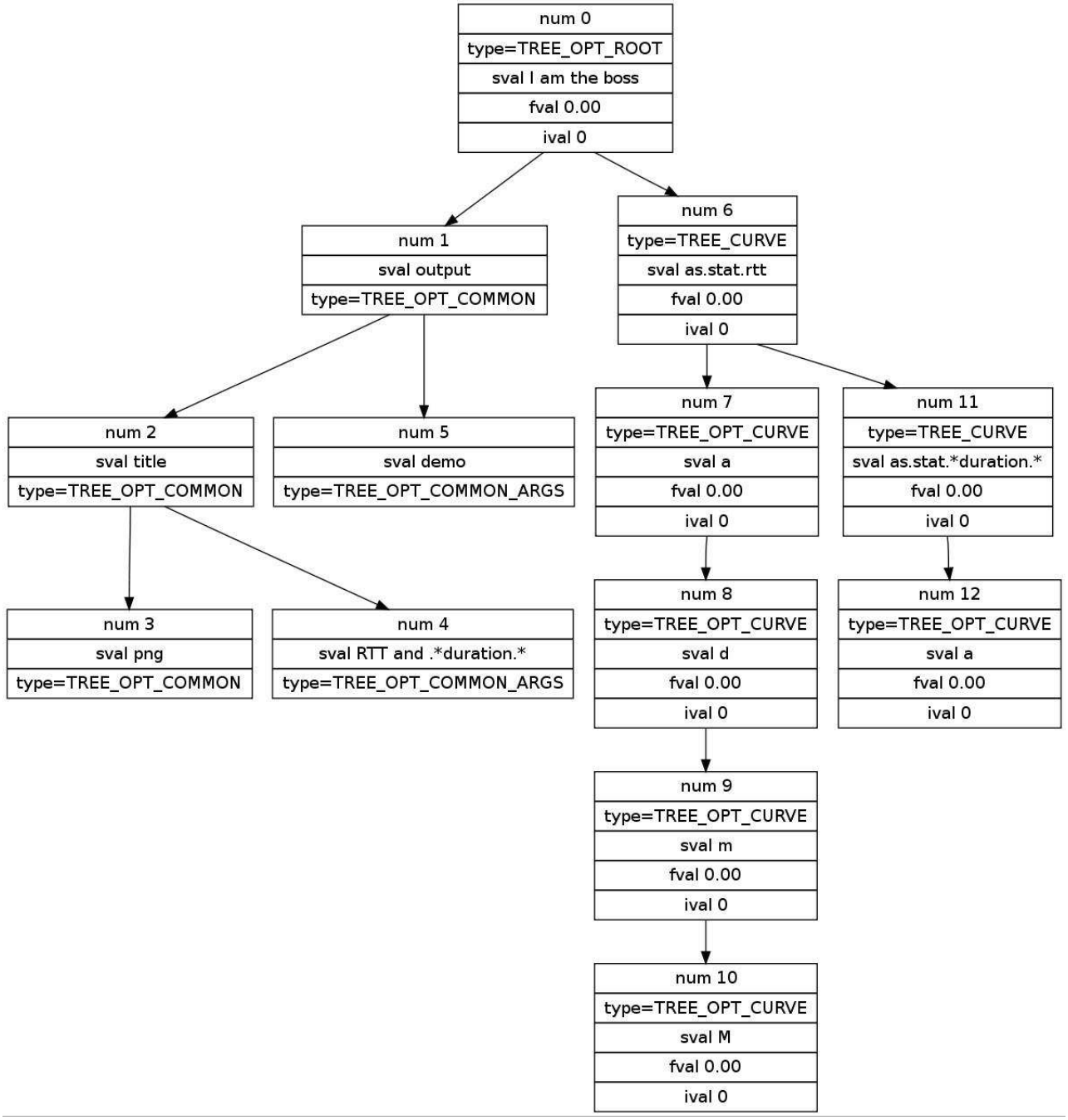


Figure 8.11 – Etape 1. opttreeRegex L'arbre syntaxique issu de la commande présentée sur la figure 8.6.

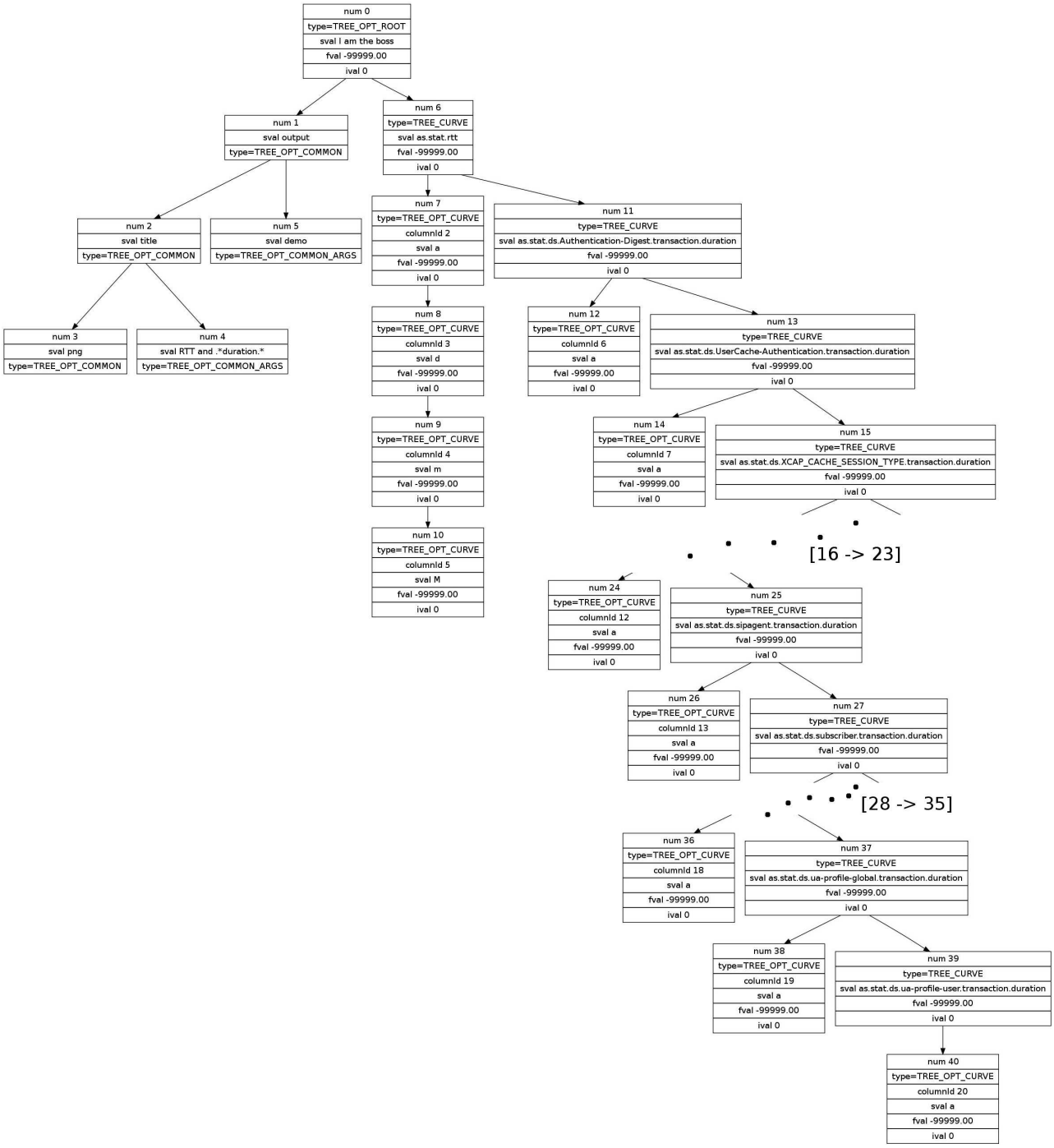


Figure 8.12 – Etape 2. L'arbre syntaxique issu de la commande présentée sur la figure 8.6.

comportement du thread pool (voir la section 2.2.2), des transactions distribuées (voir la section 2.2.3) et du réacteur. Sa qualité essentielle est de pouvoir croiser et illustrer des informations de monitoring de la plateforme.

8.5 Une partie du code source de Meter2gnuplot

8.5.1 La méthode qui crée le fichier de description de données à partir de l'arbre syntaxique

```
void printDEM(OPTTREE root,struct output_t out){
  OPTTREE aNode;
  OPTTREE mNode;
  OPTTREE MNode;
  OPTTREE dNode;
  OPTTREE opt;
  int j=0;
  if(root!=NULL) {
    switch(root->typeNode) {
    case TREE_OPT_ROOT :
      opt=searchNode(root,TITLE_OPT,TREE_OPT_COMMON);
      if((opt!=NULL)&&(strcmp(opt->sval,TITLE_OPT)==0)&&(opt->right!=NULL)&&(opt->right->
          sval!=NULL)){
        fprintf(out.out,"set title \"%s\"\n",opt->right->sval);
      }else{
        fprintf(out.out,"set title \"%s\"\n","title");
      }
      fprintf(out.out,"set style data lines\nset xlabel \"Time  (sec)\"\nset key below\
          nset style fill solid 0.2 noborder\n");
      printDEM(root->left,out);
      printDEM(root->right,out);
      fprintf(out.out,"\n");
      return;
      break;
    case TREE_OPT_COMMON :
      printDEM(root->left,out);
      return;
      break;
    case TREE_OPT_CURVE  :
      break;
    case TREE_CURVE :
      aNode=  searchNode(root->left,a_OPT,TREE_OPT_CURVE);
      mNode=  searchNode(root->left,m_OPT,TREE_OPT_CURVE);
      MNode=  searchNode(root->left,M_OPT,TREE_OPT_CURVE);
      dNode=searchNode(root->left,d_OPT,TREE_OPT_CURVE);

      int aBool = aNode!=NULL;
      int minBool = (mNode!=NULL)&&(MNode==NULL);
      int maxBool = (MNode!=NULL)&&(mNode==NULL);
      int devBool = (dNode!=NULL)&&(aNode!=NULL);
      int fillBool = (MNode!=NULL)&&(mNode!=NULL);

      if(aBool)j++;//printf("av+ j=%i ",j);//average
      if(minBool)j++;//printf("m+ j=%i ",j);//minimum
      if(maxBool)j++;//printf("M+ j=%i ",j);//maximum
      if(devBool)j++;//printf("dev+ j=%i ",j);//deviation
      if(fillBool)j++;//printf("fill+ j=%i ",j);//fill between min and max

      if ((aNode!=NULL&&(aNode->father->father->father==NULL))||
```

```
      (mNode!=NULL&&(mNode->father->father->father==NULL))||
      (MNode!=NULL&&(MNode->father->father->father==NULL))||
      (dNode!=NULL&&(dNode->father->father->father==NULL))
      )
    {
      fprintf(out.out,"plot");
    }
  //fill between min and max
  if(fillBool){
    if(strcmp(out.fileName,"-")==0){
      fprintf(out.out,"\"%s\" u 1:%i:%i title \"min_max_%s\" w filledcurves ",out.
          fileName,mNode->columnId,MNode->columnId,root->sval); //notitle
    }else{
      fprintf(out.out,"\"%s.dat\" u 1:%i:%i title \"min_max_%s\" w filledcurves ",out
          .fileName,mNode->columnId,MNode->columnId,root->sval);
    }
    j--;
    //printf("fill- j=%i ",j);
    if((root->right!=NULL)||(j>0)){
      fprintf(out.out,",");
    }
  }
  //average
  if(aBool){
    if(strcmp(out.fileName,"-")==0){
    fprintf(out.out,"\"%s\" u 1:%i title \"mean_%s\" w l lw 3",out.fileName,aNode->
        columnId,root->sval);
    }else{
    fprintf(out.out,"\"%s.dat\" u 1:%i title \"mean_%s\" w l lw 3",out.fileName,aNode
        ->columnId,root->sval);
    }
    j--;
    //printf("av- j=%i ",j);
    if((root->right!=NULL)||(j>0)){
      fprintf(out.out,",");
    }
  }
  //minimum
  if(minBool){
    if(strcmp(out.fileName,"-")==0){
      fprintf(out.out,"\"%s\" u 1:%i title \"min_%s\" w l lw 3",out.fileName,mNode->
          columnId,root->sval);
    }else{
      fprintf(out.out,"\"%s.dat\" u 1:%i title \"min_%s\" w l lw 3",out.fileName,
          mNode->columnId,root->sval);
    }
    j--;
    //printf("m- j=%i ",j);
    if((root->right!=NULL)||(j>0)){
      fprintf(out.out,",");
    }
  }
  //maximum
  if(maxBool){
    if(strcmp(out.fileName,"-")==0){
      fprintf(out.out,"\"%s\" u 1:%i title \"max_%s\" w l lw 3",out.fileName,MNode->
          columnId,root->sval);
    }else{
```

```
            fprintf(out.out,"\"%s.dat\" u 1:%i title \"max_%s\" w l lw 3",out.fileName,
                MNode->columnId,root->sval);
          }
            j--;
          //printf("M- j=%i ",j);
          if((root->right!=NULL)||(j>0)){
            fprintf(out.out,",");
          }
        }
        if(devBool){
          if(strcmp(out.fileName,"-")==0){
            fprintf(out.out,"\"%s\" u 1:%i:%i title \"dev_%s\"w errorbars lw 1",out.
                fileName,aNode->columnId,dNode->columnId,root->sval);
          }else{
            fprintf(out.out,"\"%s.dat\" u 1:%i:%i title \"dev_%s\"w errorbars lw 1",out.
                fileName,aNode->columnId,dNode->columnId,root->sval);
          }
          j--;
          //printf("dev- j=%i ",j);
          if((root->right!=NULL)||(j>0)){
            fprintf(out.out,",");
          }
        }
        printDEM(root->right,out);
        return;
        break;
      case TREE_OPT_COMMON_ARGS :
        break;
      default :
        fprintf(out.out,"error, unknown node\n");
        exit(1);
      }
    }
    else{
      return;
    }
}
```

8.5.2 La méthode qui crée le fichier de données .dat

```
void parse (OPTTREE root,struct output_t output) {

  int startHour, startMn, startSec, startDay=-1;
  int stopPLotAtyear,stopPLotAtmonth,stopPLotAtday, stopPLotAthour,stopPLotAtmn,
      stopPLotAtsec=-1;
  int elapsedTime=0;
  int oldElapsedTime=0;
  int year, month, day, hour, mn, sec, msec=-1;
  char logName[1024];
  char logLevel[1024];
  char name[1024];
  char type[1024];
  float v1, v2, min1, min2, max1, max2=0;
  float m1, m2, d1, d2=0;
  char* ln=NULL;

  OPTTREE stopPlotDateNode = searchNode(root,STOP_DATE_OPT,TREE_OPT_COMMON);
  int res=0;
```

```
if((stopPlotDateNode!=NULL)&&(stopPlotDateNode->right!=NULL)&&(stopPlotDateNode->right
    ->sval!=NULL)){
  char* stopDateStr =  stopPlotDateNode->right->sval;
  res=sscanf(stopDateStr,"%d-%d-%d %d:%d:%d",&stopPLotAtyear, &stopPLotAtmonth, &
      stopPLotAtday, &stopPLotAthour, &stopPLotAtmn, &stopPLotAtsec);
  if(res<=0){
    fprintf(stderr,"stop plot date doesn't match\n");
  }
}

startDay=startHour = startMn = startSec =-1;

while(1) {
  int level;
  int res;

  ln = giveNextLine();
  if (ln == NULL) return;
  res = sscanf(ln, "%d-%d-%d %d:%d:%d,%d %s %s %s - %s Value=%f/%f; Mean=%f/%f;
      Deviation=%f/%f; Min=%f/%f; Max=%f/%f",
              &year, &month, &day,
              &hour, &mn, &sec, &msec,
              logName, logLevel, name,
              type,
              &v1, &v2,
              &m1,&m2,
              &d1,&d2, &min1, &min2,
              &max1, &max2);
  if (res != 21) {
    res = sscanf(ln, "<%d>%d/%d/%d %d:%d:%d:%d %s %s %s - %s Value=%f/%f; Mean=%f/%f;
        Deviation=%f/%f; Min=%f/%f; Max=%f/%f",
                &level, &year, &month, &day,
                &hour, &mn, &sec, &msec, logName, logLevel, name, type, &v1, &v2, &m1
                    ,&m2, &d1,&d2, &min1, &min2, &max1, &max2);
    if (res != 22) {
      // not an Metering line, continue
      continue;
    }
  }
  //check if we have done with plotting
  if((stopPLotAtyear>=year)&&(stopPLotAtmonth>=month)&&(stopPLotAtday>=day)&&(
      stopPLotAthour>=hour)&&(stopPLotAtmn>=mn)&&(stopPLotAtsec>=sec)){
    return;
  }

  if (startHour < 0) {
  } else{
    oldElapsedTime=elapsedTime;
    elapsedTime =    (day-startDay)*24*60*60    +    (hour-startHour)*3600    +    (mn-
        startMn)*60    +    (sec-startSec);
  }
  // check if we have done accumulating log line for a period in the same second.
  if (oldElapsedTime != elapsedTime) {
    //PRINT TO OUTPUT
    fprintf(output.out,"%i ",oldElapsedTime);
    printData(root,output);
    //computeStat();//test
    fprintf(output.out,"\n");
```

```
      fflush(stdout);
      oldElapsedTime = elapsedTime;
      //sleep(1);
    }
    //   accumulation : check if the current line is one of our meters
    //FIND CURVE
    //FIND OPTIONS
    //SET VALUE FOR EACH
    OPTTREE curve=NULL;
    curve = searchNode(root->right,name,TREE_CURVE);
    if(curve!=NULL) {

    if (startHour < 0) {
      startHour = hour;
      startMn = mn;
      startSec = sec;
      startDay = day;
      oldElapsedTime=elapsedTime = (day-startDay)*24*60*60    +   (hour-startHour)*3600
            +   (mn-startMn)*60   +   sec-startSec;
    }

      OPTTREE opt=NULL;
      opt=searchNode(curve->left,a_OPT,TREE_OPT_CURVE);
      if(opt!=NULL){
        opt->fval=m1;
      }
      opt = searchNode(curve->left,d_OPT,TREE_OPT_CURVE);
      if(opt!=NULL)
        {
          opt->fval=d1;
        }
      opt = searchNode(curve->left,m_OPT,TREE_OPT_CURVE);
      if(opt!=NULL)
        {
          opt->fval=min1;
        }
      opt = searchNode(curve->left,M_OPT,TREE_OPT_CURVE);
      if(opt!=NULL){
        opt->fval=max1;
      }
    }
  }
}
```